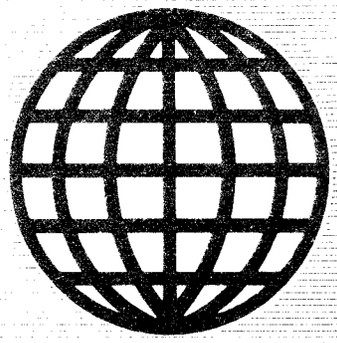


The COMPUTER JOURNAL

Programming - User Support

Applications

Issue Number 45 July / August 1990 $3.95

Embedded Systems for the Tenderfoot

The Z-System Corner

The Z-System and Turbo Pascal

Z80 Communications Gateway

Advanced CP/M

Animation with Turbo C

Real Computing

The Computer Corner

ISSN 0823-3535

The Computer Journal

Editor/Publisher
Art Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Technology Resources, 190 Sullivan Crossroad, Columbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1990 by Technology Resources.

Subscription rates—$18 one year (6 issues), or $32 two years (12 issues) in the U.S., $24 one year surface in other countries. Inquire for air rates. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used trademarks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder II, Dos Disk; Plu*Perfect Systems. Clipper, Nantucket; Nantucket, Inc. dBase, dBASE II, dBASE III, dBASE III Plus, dBASE IV; Ashton-Tate, Inc. MBASIC, MS-DOS, Windows, Word; MicroSoft. WordStar; MicroPro International. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C, Paradox; Borland International. HD64180; Hitachi America, Ltd. SB180; Micromint, Inc.

Where these and other terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

The COMPUTER JOURNAL

Issue Number 45 July / August1990

Editorial 3

Letters 4

Embedded Systems for the Tenderfoot 5
Getting started with the 8031.
By Tim McDonough.

The Z-System Corner 8
A continuation of using scripts with MEX.
By Jay Sage.

The Z-System and Turbo Pascal 16
Patching TURBO.COM to access the Z-System.
By Joe Wright.

Embedded Applications 20
Designing a Z80 RS-232 communications gateway, part 1.
By Art Carlson.

Advanced CP/M 24
String searches and tuning JetFind.
By Bridger Mitchell.

Animation with Turbo C 27
Part 2, screen interactions.
By Clem Pepper.

Real Computing 34
The NS32000.
By Richard Rodman.

The Computer Corner 40
By Bill Kibler.

ERAC CO.

P.O. Box 1108, 14179 Halper Road
Poway, California 92064 · (619) 679-8360

Baby 386-20/24 Motherboard

One 32 bit-slot, Five 16-bit slot, Two 8-bit slot. Maximum 8 meg on board (1 meg SIMMS). 0K on board. 80287 or 80387 or Weitec Co-Processor slot. Shadow RAM.

-20 $630 -24 $650

1 meg SIMMS-80ns $115

16 BIT VGA Card

RESOLUTION	COLOR
640x480	256 of 256*
800x600	16 of 256*
1024x768	16 of 256*

*with 512K RAM

Runs all standard VGA modes, 256K RAM on board. Drivers for AutoCad, Lotus, Framework, GEM, VP, WP, WS & MS Windows. 1 year warranty.

For 512K $30 extra $185

MOTHERBOARDS 386DX/SX/286

DX5000 386/25MHz-32K Cache-16 Meg-Simms-AMI Bios-Full $996
DX4000 386/20/24MHz Shadow-8 Meg-Simms-AMI-Baby 630/650
P9400 386SX/16MHz 8 Meg-Simms-Phoenix-Bios-Baby 375
P9200 386 SX/16MHz 2 Meg-Dip-AMI-Bios-Baby 365
SUPABOARD 286/12MHz 4 Meg-Dip-Dallas Clk-AMI-Baby 197

I/O, KB, CONTROLLERS, Etc.

XT Enhanced Hard Disk Drive Controller with Cables 48
DC-11M AT Hard/Floppy Contr. 1:1 438Kb/Sec, To 2048 Cyl. 93
AT I/O PLUS 1 Par(Lpt 1-3), 2 Ser(Com 1-4), Game, Cables, Ser 2(opt) 38
VGA-16 16-bit VGA Board 800x600 with Driver Software 118
2400 Baud External Modem, with Software and Manual 99
KB5161 AT/XT 101 Keyboard, Cherry Keyswitches (Click) 47
COLORMOUSE Black-Red-Blue-Beige-Green-Yellow, Software & Manual 39
VOICE MASTER KEY, Add voice commands to software XT/AT 147

MONITORS

VGA 1489 1024x768, .28 Dot, 14", Swivel, Hor. 31/35KHz, Vert. 50, 60, 70, 87 Hz 430

CASES

MINI-TOWER, 230 W, Reset, Turbo, Keylock, Speaker 143
TOWER, 230W, Reset, Turbo (2 Dgt), Keylock, Speaker 229

KAYPRO Equipment Bargains

9" Green Monitor-83, 84, K16 ..... $60
Host Interface Board 15
Keyboard 50
Replacement Power Supply 70
Drivetek 2.6M FDD (Robie or K4X) 75

We Repair CPM Kaypros

CPM COMPUTERS

K4-84 425
K10 495
K4X 425

We carry all IC's for Kaypro repair.

IC's

81-189 Video Pal $15
81-194 RAM Pal 15
81-Series Char. Gen. ROMs 10
81-Series Monitor ROMs 10

Test Equipment

OSCILLOSCOPES

TEK 7403N/7A18N/7B50A 60 MHz ... $650
Leader LB0520 30 meg Dual Trace 300
TEK 475 Dual Trace 200 MHz 1250
Scope Probe x1, x10 100 MHz 25

ANALYZERS

TEK 491 10MHz-40GHz $3500
HP851B/8551B 10HMz-40GHz 1500
Biomation 805 Waveform Rcrdr 195
Biomation 8100 2-Channel Waveform Recorder 295
HP1600A Logic Analyzer 16ch 295
HP1600A/1607A Logic Analyzer 32ch .. 495
Gould K40 32ch Logic Analyzer 750

MISCELLANEOUS

Optronics 550 MHz Freq Cntr $95
Heatgun 120Vac 7 A 35

TERMINALS

Televideo 925 ..... $99.00

NICds

AA Cells .6ah $1.00
12V Pack AA Cells .7ah 6.50
Sub-C Cells 1.5ah 1.50
12V Pack Sub-C 10.00
Double D Cell 2.5V 4ah unused 8.00
C Cells 1.75
7.2V RC-Pack 1.2ah 18.00

GEL CELLS

6V 5ah 5.00
6V 8ah 6.00
12V 15ah 15.00
12V 2.5ah 8.50
D Cell 2.5ah 2.00

ROBOTICS

5V DC Gear Motor w/Tach 1"x2" $7.50
Z80 Controller with 8-bit A/D 15.00
12V Gear Motor 30 RPM 7.50
Cable: DB9M-DB9F 1' length 2.00
High Voltage Power Supply
Input: 15-30V DC
Output: 100V 400V 16KV 6.50

SWITCHERS

AT 200W Pulls, tested $35.00
5V/75, 12V/6, -12V/3, -5V/5 ..... 85.00
5V/9.5A, 12V/3.8A, -12V/.8A 39.00
5V/3A, 12V/2A, -12V/.4A 19.50
5V/6A, 12V/2A, -12V/1A 29.00
5V/6A, 24V/1 1/4A, 12V/.6A, -12V/.6A .. 29.00
5V/30A 39.00
5V/100A 100.00
5V/120A 110.00
HP DC/DC 12Vin,5V/8A,12V/5A,-5V/.3A . 45.00

VERSATEC 8222F 22"

Electrostatic Printer Plotter
200 dots per inch. Up to D size.
1" per second $2,999

AT 80286-6 CPU BOARD

with reset and mono/color switch. Connector for KB, Battery & SPKR. Phoenix Bios (tested with Award 3.03), 6MHz, can be upgraded to 8 or 10MHz. Use with backplane, add memory board, I/O board, etc.
ONLY $99

(619) 679-8360

MINIMUM ORDER $25.00
TERMS: VISA OR MasterCard (Add 3%). Certified Checks, Money Order. NO COD Personal Checks must clear BEFORE we ship. Include shipping charges. California residents add 7 1/4% Sales Tax

Editor's Page

Do You Remember...

I've been cleaning up and rearranging, and kept stumbling over a pile of Byte magazines. I dropped the subscription two years ago, but still had a seven year accumulation. I decided that most of the information was outdated and that it was time to clip what I wanted and discard the rest.

It was interesting to note that I saved more from the older issues than I did from the more recent ones. I clipped *Ciarcia's Circuit Cellar* from every issue – most issues that was the only thing worth saving. Byte made a serious error when they lost Ciarcia, but it was to our advantage because he started his own magazine, *Circuit Cellar INK*. You should definitely subscribe to it if you read his articles in Byte or if you enjoy hardware projects. Contact them at Circuit Cellar INK, 4 Park Street, Suite 20, Vernon, CT 06066, (203) 875-2751.

The clipping process took much longer than I expected because I got sidetracked by ads. Ads for Apple II and CP/M reminded me of where we were a few years ago. Then there were ads for CP/M86 and the p-System (anyone else remember that?) Other familiar names were Morrow, CompuPro, Sinclair, TeleVideo, North-Star, Commodore, Tarbell, Cromemco, Osborne, and Victor.

I had forgotten how much change had occurred during the past decade, until I saw history flash before me as I thumbed through the magazines. It made me wonder if the changes during this decade will be as dramatic as those in the past decade. In the year 2000, which names will make us say "Do you remember..?"

TCJ's Toolkit

People frequently ask us what tools we use, as though we had all the answers. While we don't have all the answers, we do have the opportunity to examine a lot of products before selecting the ones we will work with.

If anyone tells you that they are completely objective in selecting tools, don't believe them. We all have our own personal preferences and are very subjective when making decisions. You and I have different methods of working, and different priorities. Some of us like to participate and take part in what we are doing while others want to stay at a distance and not become involved. Some are impressed by flashy colors and lots of chrome while others look for rock solid performance.

Everyone wants the most that they can obtain for their money, but there is a difference between value and cheap. I once saved a few dollars when replacing a drive motor on a production line conveyor belt, only to have the *cheap* motor fail a few days later. About 50 production line workers sat around smoking and drinking pop (and collecting full pay!) while I struggled to install a better motor. While I was doing this my boss kept telling me how much the *cheap* motor was costing in lost wages, and explained the difference between *cheap* and *value*. I look for tools which have good value, but cheap ones are worthless if they fail to do their job. As one rancher explained, "If you want first class oats, you pay a first class price. However, if you'll settle for oats that been thru the horse once, they come a lot cheaper."

To make or to buy is another decision which must be faced in many situations. I don't think that anyone would try to build a disk drive, unless they were an inventor attempting to develop something new. That decision is easy. But, the answer is not as apparent for things such as EPROM burners, EPROM erasers, ROM emulators, development boards, etc. On these we each have to make our own decision. Sometimes we'll decide to build something in order to learn how to do it.

In addition to value and the correct quality for the applications, I prefer to deal with products having the greatest portion performed in the U.S. I realize that many of the chips are made overseas, but with most of the tools I use the boards are designed, made, and stuffed here. In addition to supporting American products, I can talk to the developers if I need support.

Some of the tools we use are as follows:

8051 Cross-Assembler and Simulator – PseudoCorp, 716 Thimble Shoals Blvd., Suite E, Newport News, VA 23606, Phone (804) 873-1947 (see their ad in this issue). Dave Akey has a broad line of macro cross-assemblers, simulators, and cross-disassemblers which all run on an IBM PC. His products were recommended to me by Tim McDonough from Cottage Resources, and they work very well.

8031 Development Board/uController Module – Cottage Resources Corporation, Suite 3-672, 1405 Stevenson Drive, Springfield, IL 62703, Phone (217) 529-7679. Tim McDonough (see his article and ad in this issue) has developed these boards as a result of what he would liked to have had while developing embedded applications. These were designed to fill a need, by someone who was working in the area – not something designed by a committee! I'm working with the Control-R I board which is a bargain at only $39.95. I had been trying to get something going using a solderless breadboard, but it didn't work because of poor connections. Now, with the Control-R I, I can concentrate on the application instead of fighting with the breadboard problems. Highly recommended for experimenting, prototyping, and the core for small quantity applications.

EPROM PROGRAMMER – Needham's Electronics, 4535 Orange Grove Ave., Sacramento, CA 95841, Phone (916) 924-8037. Alan Needham produces a quality product at a very reasonable price (see their ad in this issue). I am using the PC internal card version, but they also have stand-alone versions for production use. It automatically sets programming voltage, uses intelligent programming algorithm, and can split the bytes by two or four for 16 and 32 bit systems. I have heard many horror stories about the overseas card bases programmers, but Needham's is made in the US, is well made, and is well worth the $139.95.

EPROM ERASER – Ultra Violet Products, Inc., 5100 Walnut Grove Ave., San Gabriel, CA 91778. Their DE-4 eight chip eraser (available direct or from Jameco for $69.95) has an intensity of 6800 uW/square cm. It is well built and a good choice for mid-level use. They also have larger units for production use.

BAR CODE READER – Adaptive Technologies, 810-208 Los Vallecitos, San Marcos, CA 92069, Phone (619) 744-8087. Douglas Johnson has just released his FlexScan I decoder, which includes a PC card, wand, and software. It can be accessed as either a keyboard wedge, or driven directly thru the API (Application Program Interface). We are basing a bar code tutorial on this product. You'll see a lot more bar code applications, and you should start becoming familiar with the technology. Send for their literature.

(Continued on page 35)

8031 µController Modules

NEW!!!

Control–R II

√ Industry Standard 8–bit 8031 CPU
√ 128 bytes RAM / 8 K of EPROM
√ Socket for 8 Kbytes of Static RAM
√ 11.0592 MHz Operation
√ 14/16 bits of parallel I/O plus access to address, data and control signals on standard headers.
√ MAX232 Serial I/O (optional)
√ +5 volt single supply operation
√ Compact 3.50" x 4.5" size
√ Assembled & Tested, not a kit

$64.95 each

Control–R I

√ Industry Standard 8–bit 8031 CPU
√ 128 bytes RAM / 8K EPROM
√ 11.0592 MHz Operation
√ 14/16 bits of parallel I/O
√ MAX232 Serial I/O (optional)
√ +5 volt single supply operation
√ Compact 2.75" x 4.00" size
√ Assembled & Tested, not a kit

$39.95 each

Options:
- MAX232 I.C. ($6.95ea.)
- 6264 8K SRAM ($10.00ea.)

Development Software:
- PseudoSam 51 Software ($50.00) Level II MSDOS cross–assembler. Assemble 8031 code with a PC.

- PseudoMax 51 Software ($100.00) MSDOS cross–simulator. Test and debug 8031 code on your PC!

Ordering Information:
Check or Money Orders accepted. All orders add $3.00 S&H in Continental US or $6.00 for Alaska, Hawaii and Canada. Illinois residents must add 6.25% tax.

Cottage Resources Corporation
Suite 3–672, 1405 Stevenson Drive
Springfield, Illinois 62703
(217) 529-7679

Letters

Z80 Controllers

Your letter on small controllers was quite timely. A fellow hacker and I had been discussing (arguing) this very subject. He maintains that the (IBM) PC has made 8 bit systems obsolete. The world is now standardized, and all computing in the future will be done with PC-compatible architectures.

I pointed out that the vast majority of micros sold are 4- and 8-bitters. His PC had one 16-bit CPU, but half a dozen 8-bitters; in the keyboard, printer, modem, each disk drive, and in most other peripherals.

As to the future, RISC machines could be the death of the PC. They offer and order of magnitude more performance without the PC's ad hoc architecture and programming difficulties. It won't surprise me if RISC spawns a second micro revolution. They'll replace today's micros as thoroughly as they in turn replaced yesterday's minis.

Imagine a 1995 Timex/Sinclair II, with it's RISC CPU and a meg of memory bit-banging video and disk I/O while emulating any of a dozen CPUs...

I am a good hardware engineer, and have several very small Z80 systems in my portfolio. It takes very little effort to put together powerful, inexpensive systems. I'm inclined toward a Z80 with 32K RAM, 32K EPROM, and two 5380s for I/O. Material cost would be around $30.

One danger is creeping featurism. Why not a 64180? 512 K RAM. Disk interface. Fancy video. Next thing you know, it's still another do-all computer, competing with the PC. No, I thing it should be a KISS design. If you want more horsepower, add more boards, or buy a PC.

Why 5380s? For one, they have real-world drive capability. 8255s are cheaper, but need extra drivers to run anything real (LEDs, power transistors, relays, etc.). Second, they make fast inter-processor bus (SCSI) easy to implement. Why two 5380s? Because 18 I/O lines is pretty skimpy for most applications. If more than 36 lines of I/O is needed, add another board.

Software is the hardest part. If we depend on assembly language programming, it will be inaccessible to most people. If we embed a high-level language, it is sure to offend the purists. BASIC? Forth? C? All of them have hate groups that won't use them, and none are suitable for multi-processor systems.

I'd like to see something Smalltalk-like, sort of a "tiny-Talk." Each board is intelligent in its own right, and can be programmed in a high-level language. In a multi-board system, they pass messages between each other (the user/programmer is just one more source/sink for messages). Smalltalk is the only language I know that has a structure for non-proceedural non-hierarchical processing.

As a programming metaphor, I'm thinking of a spreadsheet. Each cell is an I/O port or RAM location. You FORMAT the cell to determine how it inputs or outputs data (serial, parallel, individual bits, etc.). The FORMULA is a one-liner in a calculator-like language.

For Example, format cell A1 as an 8-bit output, and B1 as an 8-bit input. The program in A1 can do some boolean manipulation, like A1=(B1+5)/2. If row A is physically handled by one board, it continuously recalculates the output for A1, which involves sending a message to the row B board to ask the current value of B1.

So I'm interested! How do you suggest we proceed?

Lee A. Hart

PC Development System

Lee Hart sent me a copy of his May 16th letter to you on small systems development and encouraged me to respond as the "fellow hacker...who maintains that the (IBM) PC has made 8-bit systems obsolete". Well...

I'm a good hardware engineer too, and I find it hard to develop and manufacture a system as cheap as an IBM PC. For $500 new, and maybe $250-$400 used, you get a 2-floppy monochrome system. Admittedly,

(Continued on page 36)

Embedded Systems for the Tenderfoot

Getting Started with the 8031

by Tim McDonough, Cottage Resources Corp.

How many times have you had a great idea for a project that required some "smarts" but were put off by either the cost or the size of dedicating a PC to controlling the system? Or maybe your latest brainstorm required battery operation to be genuinely useful or your users just wouldn't tolerate keyboards, disk drives, etc. Learning how to develop stand alone microprocessor based projects, or embedded systems as they are called in the trade, may be just the ticket to getting that project off the shelf and into the real world.

Throughout the next couple of issues, or for however long Art is willing to put up with me, I'm going to present a few projects that will give you the basics of embedding a microprocessor in your projects.

This isn't the kind of stuff that only rocket scientists and black belt computer hackers can succeed at. If you can already do some programming in BASIC, Pascal, C, or whatever and have some rudimentary knowledge of digital electronics, then getting started with embedded systems is within your reach. You probably won't have NASA beating down your door to work on the next generation of space shuttles, but you will be able to develop some decent automation projects that do the job you need to accomplish.

There are a variety of processors suitable for use in embedded systems. Z80s, 80286s, 68000s, and almost any other type you can think of are used in a variety of systems. The particular chip that I'll be concentrating on is the 8031 which is manufactured by Intel and second sourced by several other manufacturers. The 8031 is a member of what Intel calls the MCS-51 family. All members of this family have similar characteristics and a varying number of features and functions depending on your particular needs. Key features of the 8031, the "bare bones" model are as follows:

- 128 bytes of internal RAM
- Built in UART for serial communications
- 2 16-bit counter/timers
- 4 8-bit Input/Output ports
- 2 external hardware interrupt lines
- bit addressable memory and I/O ports

Before I go on, let me say that the 8031 is not the only 8- bit microcontroller around; there are similar products made by Motorola and others. In any given group of engineers, scientists, or programmers, you can get into discussions akin to religion or politics if you say something like "The 8031 is the best 8-bit microcontroller on earth." Fact of the matter is, the "best" is the one that you are the most comfortable using that will get the job done. I happen to have had access to a lot of people and resources that supported the 8031 while I was a "tenderfoot" and so I continue to use what I know best.

Tim McDonough is the President of Cottage Resources Corporation. The company manufactures and distributes several single board computers based on the 8031 and is a dealer for PseudoCorp brand 8031 cross-assemblers and cross-simulators. He may be contacted at: Cottage Resources Corporation, Suite 3-672, 1405 Stevenson Drive, Springfield, Illinois 62703, (217) 529 - 7679.

The circuit shown in Figure 1 is a real bare-bones 8031 system. It contains the 8031, an address latch (more later), and 8K of EPROM to hold your application program, data tables, etc. This circuit is very similar to one presented by Steve Ciarcia in his Circuit Cellar column that appeared in the August 1988 issue of BYTE magazine.

You may wonder about the lack of RAM in the schematic. The applications I'll be presenting are written in 100% assembly language and since this gives total control of the system operation, the 128 bytes of internal RAM contained in the 8031 will be more than adequate.

Before we delve into the first application, there are a few "tools of the trade" that are required aside from the circuit shown in Figure 1. Whether you buy an off the shelf 8031 board or hand wire your own you will need the following minimum items to get into developing your embedded system:

- The Intel 8-bit Microcontroller Handbook
- A host computer on which to write software
- A cross-assembler to produce 8031 object code
- An EPROM programmer to get your object code into a blank EPROM.
- An EPROM eraser to get rid of your inevitable mistakes.

The amount of money you'll spend on these items can vary. There is no upper limit to price but a workable system such as the one I typically use at home will cost about $270.00. This includes a cross-assembler, EPROM programmer, and an EPROM eraser. If you build up a simple 8031 circuit of your own and purchase a few EPROMs for experimenting, you can still get started for about $300.00.

So much for the introduction. This first project will get you used to how some of the more useful pieces fit together and provide you with a working code example that you can use as the basis for your first experiments. It demonstrates how to read digital input sources such as switches, photocells, etc. and shows how to control simple digital output devices such as relays, LEDS, and small motors.

Before jumping into the project itself, a quick tour of the "computer" is in order. As shown in Figure 1, the system is composed of 3 Integrated Circuits – an 8031, a 74LS373 address latch,

and a 2764 EPROM.

The 8031 is essentially the entire computer. It contains RAM, several counter/timers, system clock circuitry, and a full- duplex UART that can be used to implement a serial communications port. The 2764A is an 8K EPROM that is used to hold the program that the 8031 will execute. A 74LS373 is used as a "glue" chip to hold the other parts together. The 8031 design multiplexes the low order address lines (A0-A7) onto the same 8 lines as the data (D0-D7). A separate external Address Latch Enable (ALE) line (Pin 30) is used to latch the lower byte of the 16-bit address into the 74LS373, after which the bus is used for data transfer.

The purpose of the system described here is of little interest; it's the implementation that should pique your interest. Essentially it is a complicated, but flexible equivalent of an exclusive "or" (XOR) logic gate. That is, whenever one and only one of the inputs is active (grounded), the LED will light. Any other input conditions will cause the LED to remain off as shown in the following truth table:

```
TRUTH TABLE FOR A TWO INPUT ''XOR'' GATE

INPUT A (P1.0) | INPUT B (P1.1) | OUTPUT (P1.2)
---------------+----------------+--------------
     HI        |       HI       |      HI
---------------+----------------+--------------
     HI        |       LOW      |      LOW
---------------+----------------+--------------
     LOW       |       HI       |      LOW
---------------+----------------+--------------
     LOW       |       LOW      |      HI

 HI == +5VDC, LOW = 0VDC/GND
```

This software "gate" could of course be a part of a much larger system and in fact I'll be expanding on it in the next several issues. For now, making it work will give you a good opportunity to get comfortable with the basic 8031 circuit, cross-assembler, and EPROM programmer. The test circuit is shown in Figure 1. Switches A and B let you simulate the inputs to the "gate" and LED #1 will let you track the output.

Remember I mentioned earlier that a lot of things don't require much in the way of memory? The code for the exclusive-OR gate uses none of the 8031's RAM and only occupies 19 bytes of the system's 8K EPROM. The source code is shown in Listing 1.

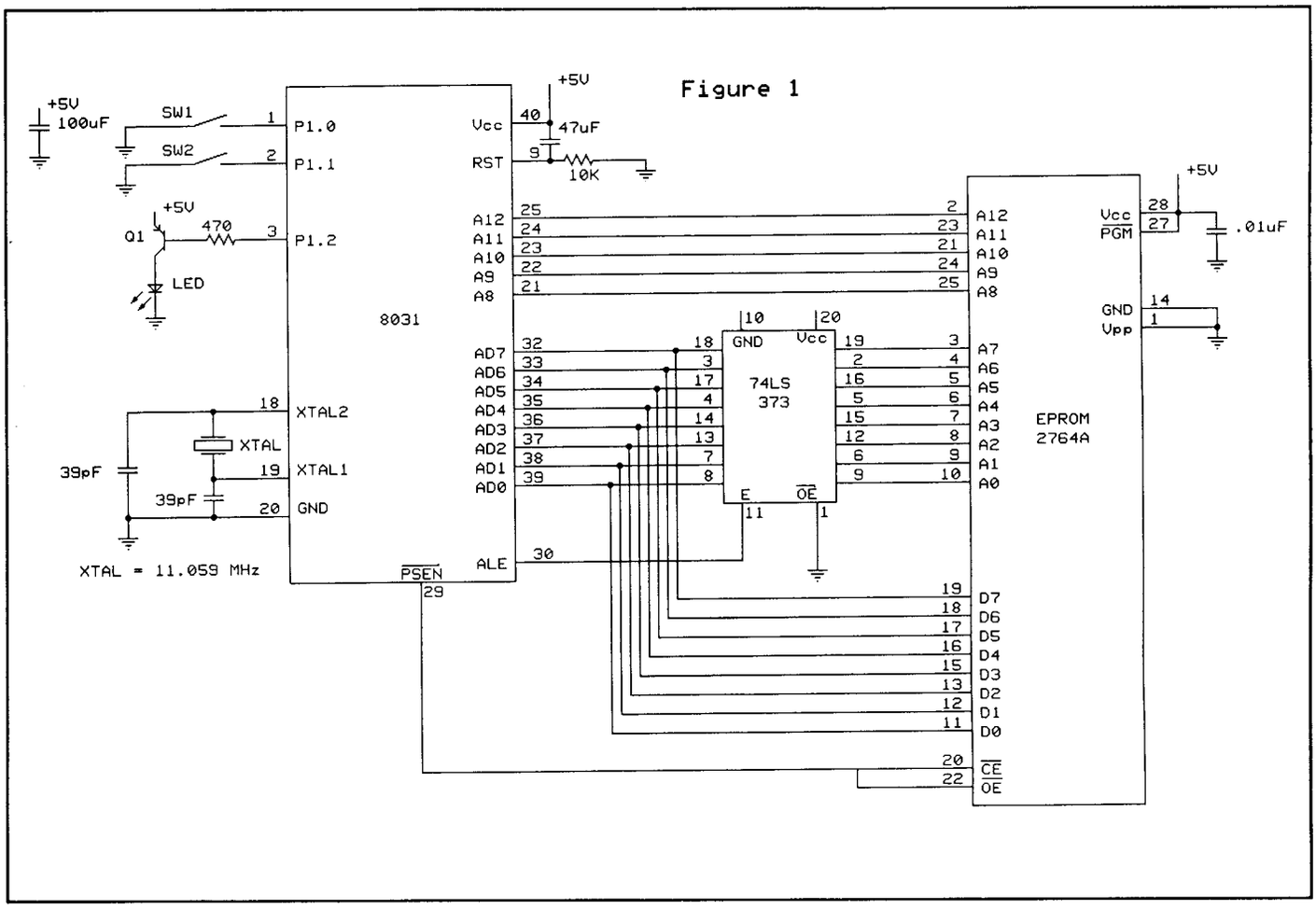
Coding in assembly language, although at times a bit more tedious, is no different than using whatever high level language you're using now. In the case of implementing the exclusive-OR gate in software I simply check each possible set of conditions one by one. The algorithm, in pseudo code, is as follows:

```
"main"
if SW1 is set then
   goto "check SW2"
if SW2 is set then
   goto "turn LED on"
goto "turn LED off"

"check SW2"
if SW2 is clear then
   goto "turn LED on"

"turn LED off"
clear LED1
goto "main"

"turn LED on"

set LED1
goto "main"
```

The actual assembly language source code isn't quite as lucid as

LISTING #1 -- Embedded Systems for the Tenderfoot

```
000001 0000                    ; XOR.ASM -- May 9, 1990
000002 0000                    ;
000003 0000                    ; (C) Copyright 1990 by Tim McDonough
000004 0000                    ; Cottage Resources Corporation
000005 0000                    ; Suite 3-672, 1405 Stevenson Drive
000006 0000                    ; Springfield, Illinois 62703
000007 0000                    ; (217) 529 - 7679
000008 0000
000009 0000                              .ORG  H'0000      ;ASSEMBLE TO BEGIN AT 0000 HEX
000010 0000
000011 0000                    ; EQUATES are used to give memory locations, registers and bits somewhat
000012 0000                    ; english-like names
000013 0000
000014 0090                              .EQU  SW1,P1.0    ;SWITCH #1 ON PORT 1, BIT 0
000015 0091                              .EQU  SW2,P1.1    ;SWITCH #2 ON PORT 1, BIT 1
000016 0092                              .EQU  LED,P1.2    ;LED #1 ON PORT 1, BIT 2
000017 0000
000018 0000                    ; The main program is an endless loop that constantly compares the status
000019 0000                    ; of the two input pins against the XOR truth table and adjusts the state
000020 0000                    ; of the output pin accordingly.
000021 0000
000022 0000 309005             START:    JNB   SW1,S1_0    ;IF S1 = 0 CHECK S2
000023 0003 309109                       JNB   SW2,LED_ON  ;S1 = 1 AND S2 = 0
000024 0006 8003                         SJMP  LED_OFF     ;S1 = 1 AND S2 = 1
000025 0008
000026 0008 209104             S1_0:     JB    SW2,LED_ON  ;S1 = 0 AND S2 = 1
000027 000B
000028 000B D292               LED_OFF:  SETB  LED
000029 000D 80F1                         SJMP  START
000030 000F
000031 000F C292               LED_ON:   CLR   LED
000032 0011 80ED                         SJMP  START
000033 0013                              .END
```



the pseudo code. "XOR.ASM" is shown in Listing 1. The file is formatted for the PseudoSam 51 Symbolic Assembler (See the notes at the end of the article for details).

When the 8031 is first powered up, it starts trying to execute instructions found at location 0000 Hexadecimal. The .ORG directive tells the assembler to locate the assembled instructions beginning at 0000 Hexadecimal as well. Next an "equate" directive is used to assign some "reader friendly" names to several of the 8031's I/O pins. The first equate assigns the symbol "SW1" to Port 1, Bit 0 (P1.0), the location were I've attached Switch #1.

A major advantage of the 8031 for control applications is its ability to manipulate single bits as opposed to performing operations on only full bytes. The JNB instruction looks at the bit referenced in the first argument and causes program execution to jump to the label referenced in the second argument if the bit was not set. If the bit was set, nothing happens and the next instruction is executed. The JB instruction does just the opposite, jumping if the referenced bit is set.

The other instructions used are fairly straight forward. SETB and CLR set and clear bits, respectively. SJMP is practically like a GOTO in BASIC or FORTRAN. It simply forces program execution to continue to a certain location.

I hope it's obvious that using a microprocessor, EPROM, latch, crystal and all the other assorted components in this project as a replacement for an XOR gate isn't a great idea in and of itself. If all you need is a gate then by all means use one. But what if we wanted to transmit the status of the two switches to a PC via a serial port? What if we needed to measure a temperature in addition to knowing the two switch positions? To accomplish either of these tasks and many more, very little additional hardware is needed. It's mostly "code".

If you decide to experiment more with the 8031, the basic circuit described in this article can be used over and over for a variety of projects. Some people will want to wire-wrap or point to point wire their own board to make it exactly what they want. If you don't have the time to build or would rather spend the bulk of your time coding, an assembled and tested version of this circuit that also includes the circuitry for an RS232 compatible serial port (I'll be talking about serial port programming in an upcoming issue) is available from Cottage Resources Corporation. The Control-R I (pronounced "Controller One") is available for $39.95 and the PseudoSam 51 cross-assembler is available for $50.00 (plus $3.00 Shipping per order) from Cottage Resources Corporation, Suite 3-672, 1405 Stevenson Drive, Springfield, Illinois 62703, (217) 529-7679●

References:

Embedded Controller Handbook, Volume I 8-bit 1988, Intel Corporation.

Mastering Digital Device Control by William Houghton, 1987, SYBEX. *(Editor's Note: This book is out of print.)*

Why Microcontrollers? Part I, by Steve Ciarcia, August 1988, BYTE Magazine.

The Z-System Corner

by Jay Sage

Last time we gave you a pretty thorough presentation of the script language from the MEX-Plus telecommunications program. However, as necessary as such documentation is, it does not really teach one how to make effective use of the language. So this time we will present as a teaching example significant portions of the script suite that I use for almost all my telecommunication work. It is the most complex script that I have ever written, illustrates many techniques, and might be very useful to many of you as well.

This script is far from perfect. Every time I work with MEX-Plus I learn something more about it, and that was a large part of my motivation for writing these two columns. As usual, I hope that some astute readers will notice ways to improve on my script.

More MEX Commands

Before getting into the script itself, I have a few items to add to the discussion from last time. First, I forgot to mention one extremely important MEX command; second, just today I discovered some more undocumented commands that appear to be quite interesting.

The WAIT Command

The WAIT command is one of MEX's most important commands. It allows MEX to monitor the character stream coming back from the remote system and to take various actions depending on what it sees. There are four variants of the command: WAIT DATE, WAIT TIME, WAIT SILENCE, and WAIT STRING.

The first two forms cause the script to pause until a specified date or time arrives. Obviously, you must have a real-time clock and a MEX clock module installed for these commands to work. The command forms are:

```
WAIT DATE mm/dd
WAIT TIME hh:mm
```

These commands would be useful for a script to automatically place a call during the middle of the night when phone rates are lower.

The WAIT SILENCE command waits until no characters have been received from the modem for a specified time interval. This is one way to infer that the remote system has finished what it was doing and is ready for a command from you. The syntax is:

```
WAIT SILENCE [time]
```

The most powerful of the WAIT commands is WAIT STRING, whose full syntax is:

```
WAIT STRING [time] string1 [string2 string3 string4]
```

This command takes from one to four string expressions and an optional wait time, which otherwise defaults to values set by STAT parameters. The command terminates as soon as one of the strings is detected or the time limit expires. The VALUE variable tells you the result. It will be 0 if the time limit was reached or 1, 2, 3, or 4 depending on which string was matched. You will see a number of examples of the use of this command in my PC-Pursuit script.

Undocumented Commands

MEX has quite a number of undocumented commands. These can be discovered by doing a memory dump of MEX.COM and looking for the command dispatch table. Scanning programs' command tables to see what goodies might have been built into them that the authors—for one reason or another—decided not to tell you about is a great sport. I will mention only a few of the MEX commands I discovered this way.

First, there are some commands that are just alternate names for documented functions. For example, there is a RENAME command that vectors to the same code that the REN command does.

Since, as I mentioned last time, there seemed to be a paucity of ways to get out of MEX (only about six commands!), I was quite relieved to discover the command ABORTMEX, which appears to offer yet another way! Actually, I have a recollection of having seen that command somewhere in the documentation, but it is not listed in the index and I cannot find it again. From examining the dispatch vectors, I can tell that ABORTMEX is not the same thing as CPM, EXIT, QUIT, and so on, but it seems to do the same thing.

One command that I think will prove quite useful is the PAUSE command. Its syntax appears to be like that of the undocumented PRINT. Whatever text comes after it is echoed to the screen, and the script then pauses until any key is pressed.

The MEM command looked as though it was going to be quite useful, as it displays the status of MEX's memory buffer. The trouble is, I have not been able to figure out what buffer this is! I started a capture buffer, and MEM still showed the same values and reported that none of the buffer was in use. Then I put the command in a script file, thinking it might report the status of the script buffer. Alas, the report was still the same. Perhaps this is just a command that was never fully coded. All I can say is that the buffer size reported does depend on the size of one's TPA.

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER point-and-shoot shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC-Pursuit, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Jay is now also the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail or chatted with live at the Wednesday real-time conferences (10pm Eastern time).

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog computation to solve problems in signal, image, and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE@LL.LL.MIT.EDU.

Another very interesting command is WIN. Its name suggested that it created some kind of window, and indeed it does. It is probably not documented because it does not seem to work completely correctly. I entered the command

```
WIN 5 5 12 75
```

and MEX drew a partial box of the sort that "BOX 5 5 12 75" would have and then put its prompt at the upper left corner of the window. After that, screen output was restricted to the lines in the window, but the lateral limits of the window were not observed; text still ran across the full width of the screen. The STAT command would fill just the window and then wait for a keypress to continue. Of course, I did my tests from the command line, and WIN may act differently if invoked from a script file.

The "@" command cursor addressing could still take one anywhere on the screen. Thus, it looks as though the WIN command might be useful in some special cases where one wants to keep certain status information on the screen. The window could be set to the last 20 lines on the screen, and status information could be written using "@ SAY" to the regions outside the window.

Another command whose function I thought I could guess was FLUSH. I assumed that it flushes the contents of a buffer, perhaps the capture buffer. However, I tried it with a capture buffer, and nothing seemed to happen. There must be something more subtle about it.

Finally, there are the commands DUPE, RESTORE, TRAP, LIB, EXEC, OVRINIT and probably a few others.

A Challenge

I will offer an unspecified prize to the user who does some detective work and sends me the most complete documentation on these undocumented commands. A free copy of the Mex-Pack terminal emulation and remote operation modules might be a fitting prize. Or perhaps a copy of the new ZMATE macro text editor.

I also have some recollection that someone once figured out a way to use either the strings assigned to keys or the names in the phone directory as string variables. So far, however, I have not been able to figure out how to do it. We'll include that and any other undocumented information about MEX-Plus within the framework of this challenge.

Now let's begin the look at the PCP script.

The PC-Pursuit Script

The Purpose

The purpose of this suite of MEX scripts is to make life with PC-Pursuit easier. Before the recent changes in policy, using PCP was an enormously frustrating experience. The outdial modems were almost always busy, and it sometimes took dozens or even hundreds of tries to get connected to a desired city—each try requiring one to enter one's user ID and password. Today, with the 30-hour limit on free access, things are much, much better, but it is still handy to have a script take care of the operation automatically.

Here is basically what my script does. It calls up the local Telenet access point and issues the commands to set up the proper terminal mode. Then it negotiates a connection to the city where the selected remote access system (RAS) is located. If all the modems in that city are busy, the script can keep trying. If all the 2400 bps modems are busy, it can even automatically step down and try the 1200 bps modems. Today, one rarely fails to connect on the first try at 2400, but in the past the multiple tries and automatic stepdown were lifesavers.

Once the connection to the city has been established, the script issues the commands to put the remote modem into Racal Vadic mode and then dials the number for the RAS. In Vadic mode, the modem issues call status reports, so you know when the modem is dialing, when the phone is ringing, when the line is busy, and when the call has simply failed. The MEX script monitors these reports and responds appropriately.

Once the remote system has been reached, a very short script is initiated so that a maximum amount of memory will be free for MEX to use for its capture and file transfer buffers. The script also programs several function keys to make logging onto the RAS easier.

It would be quite easy to have the PCP script chain to a script to perform the complete login operation, but I generally prefer to do this manually. That gives me a chance to notice if there are any new bulletins or other changes in the system.

Design Philosophy

Two main principles guided the design of the script suite. First, as I mentioned last time, I made it highly modular. This makes writing the script easier and clearer but, more importantly, it overcomes memory limitations. By chaining from one script to another, only one script has to be in memory at one time. By making the last script a very small one, almost no buffer space is lost during the time the user is working on the remote system, even though a script is still in operation.

The second principle is to provide as much error checking as possible. For example, at the very beginning, the script checks to make sure that the local modem is connected, turned on, and responding. I learned to do this after trying many times to run this and other scripts with the modem turned off.

In its present form, when an error is detected, the script normally issues an explanatory message and then terminates. It would be better to provide error recovery wherever possible. For example, having discovered the PAUSE command, I might now improve the script by making it pause until the user turns the modem on and presses a key. Then the script would loop back and try again.

There are quite a few places in the script where it will retry a failed operation several times before it gives up. Sometimes PC-Pursuit just seems to go out to lunch, and I have been unable to get any response from it even with manually entered commands. In such cases, of course, there is nothing more that the script can do.

Architecture

Before talking about the detailed functions of each module in the script, I would like to describe the architecture. There are 6 modules, and their relationships are shown in Figure 1.

The central script is in the file PCPMENU.MEX. The invocation script, PCP.MEX, performs some one-time operations and then transfers control to PCPMENU. After that, control branches to other scripts but eventually returns to the menu script. It is only from PCPMENU that the script can be terminated and control returned to CP/M in a graceful fashion.

The data needed to connect to a remote system can be supplied in two ways. First, the menu displayed by PCPMENU lists many commonly called systems. When one of these is selected, control branches to PCPDATA, which loads the required data into MEX

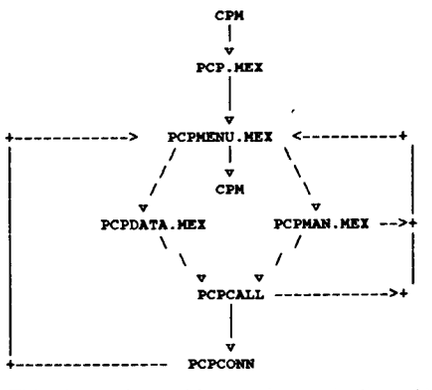


Figure 1. This shows the architectural organization of the suite of script files that comprise the complete PCP script.

numerical and string variables and function keys.

Alternatively, there is a choice for contacting a RAS that is not on the menu. In this case, the script PCPMAN (short for PCPMANUAL) provides for step-by-step, menu-driven entry of the required information. If the user decides against making that call, control can be returned to PCPMENU. Normally, however, control from either PCPDATA or PCPMAN flows to PCPCALL.

PCPCALL carries out the operations required to make PC-Pursuit connect to the requested city and then dial the requested local telephone number. When either the PCP or RAS modems is busy, the script allows the user to decide whether to continue trying and how many times. If the call cannot be completed and the user does not want to continue trying, control returns to PCPMENU.

If the remote system is reached, control is passed to the small script PCPCONN (short for PCPCONNECTED). We showed this script in the previous column. It puts the user into terminal mode for login. Whenever the user exits back to MEX command mode, a prompt is put up. Entering the command "M" returns control to PCPMENU. Other MEX commands can be entered as usual for transferring files, opening capture buffers, changing STAT parameters, and so on.

This script suite does not make use of any subroutine scripts invoked using the DO command. In all cases, control is transferred permanently to another script using the READ command. There are some subroutine command blocks defined by the PROC and ENDP commands. These subroutines are contained within the script file because they execute faster that way and because there was no reason in most cases to implement them as separate files.

We will now make a few comments about details of the individual script files. Because the code is quite lengthy, we are unable to print it all here. Many whole sections have been removed, and some comments have been cut out. However, there are several ways to obtain the complete scripts. First, they will probably be included on the ZSUS (Z-System Software Update Service) subscription disk that is released at about the time this issue appears. Second, the files will be posted on RASs. Finally, Art Carlson has offered to make them available to subscribers who send him a formatted, labeled diskette with return postage and mailer.

Script PCP.MEX

This script (Listing 1) initializes a number of variables. Note that variables that the user is likely to want to change are placed at the beginning of the script. Also note that certain values that could have been hard coded into the script, such as the default number of tries to connect to a city, are stored instead in numerical variables. This is like using EQU parameters in assembly code and is a highly recommended practice. It is not a bad maxim never to use actual numerical constants in any program; always use symbolic constants.

Table 1 shows how variables are used in the scripts. I'll have to confess that I prepared much of this list after the fact. That was a mistake. I would have made many fewer coding errors had I meticulously documented the use of variables from the very beginning. In fact, the comments next to each variable should be more extensive than what I show here. Some of the information is incomplete; some may even be wrong.

Note the way command line parameters are handled in PCP.MEX. The full syntax for invocation of the script is

```
READ PCP [menu choice] [city tries] [RAS tries]
```

There are three optional parameters. The first is a menu choice. If you know that you want to place a call to RAS number 1 on the menu, you can use the command "READ PCP 1" to do so directly. In case you think that it is too hard to remember the numbers, you are right; that's where ARUNZ aliases come in! My LADERA alias becomes "MEX READ PCP 1". The other two parameters are the default number of times to connect to a city's outdial modem and the destination modem, respectively.

The parameter values are carefully validated in the script. If the values are illegal, then the built-in defaults are used. Validating user input is something that none of us does enough of.

In the part of the script that checks the local modem, there is the command

```
WAIT STRING 2 "OK" "0"
```

This makes the script wait for up to 2 seconds for the modem to respond with either "OK" (which it will do in normal verbose mode) or "0" (which it will do if it was left in terse mode). It is always a good idea to have code anticipate and deal with all possible situations.

I have an MNP level-4 modem, and Telenet supports MNP error correction at the indial port that I use. Therefore, I put the modem into MNP mode at the beginning of the PCP script and set it back to normal mode on normal exit. I have omitted this code from the listing. If you do not have an MNP modem, you would, of course, remove (comment out) this part of the script.

The script is pretty carefully written to make sure that everything is proceeding correctly. After connecting to Telenet, up to two attempts are made to establish the required synchronization. This same technique of looping with a max-tries count in variable %z is used in many places throughout the scripts. Note the use of the SLEEP command to introduce delays when the system you are communicating with does not always respond immediately.

Script PCPMENU.MEX

The menu in this script (Listing 2) is drawn inside a box and has the RAS selections displayed in three columns. Free entries are filled in with a row of dots. We have included only enough entries to show how they are generated. It is important that free entries be trapped later in the script.

The menu is adaptive. If one is currently connected to a city, the city code and data rate are shown in the menu header, and the menu selections for changing the data rate that otherwise appear at the bottom are omitted.

Another example of robust coding is provided by the ABORT routine at the end. Whenever this script is running, we should be connected to PC-Pursuit. Therefore, the script attempts to disconnect by sending the HANGUP command that PCP likes to see. If several attempts to disconnect in this way fail, however, we simply drop carrier.

Script PCPDATA.MEX

This script (Listing 3) is quite straightforward. It sets some default key definitions that apply to many systems. Then it branches to the entry for the RAS selected from the menu in PCPMENU. Here the variables necessary to place the call are set and any other function key definitions are made. Then control is transferred to PCPCALL. In the listing we show the entry for only one RAS.

Script PCPMAN.MEX

This is the second most complex module in the series (Listing 4). I used to have a much simpler and less agreeable version; in honor of this column I just rewrote it. It used to require manual entry of all information, and it provided no checking. Now it puts up a menu of all PCP cities and allows selection by number. It also keeps track of the area codes covered by each city. When there is a second area code, a menu lets the user choose. The script is even smart enough to include the area code as part of the local number when needed and to insert the "1" prefix for those phone systems that require it.

There are two major subroutines in this module. Routine CITYNAME takes the city number and produces the name of the city and state in a string variable. Routine PCPCODE generates the PCP outdial code and the telephone area codes for the city. The same CITYNAME subroutine is also included in module PCPCALL.

Listing 1: Script PCP.MEX.

```
.. PCP.MEX: MAIN PCP SCRIPT (04/15/90)
..
.. This is the main entry point to the script for automating
.. calls via the PC-Pursuit network.  This part of the script
.. handles initialization of MEX and the modem and establishes
.. the connection to the local access number.

.. Two things we have to take care of right away

screen off
stat sep ";"

.. ------------------------------------------------------------
..
..      Configuration Information
..
.. ------------------------------------------------------------

phone pcp=000-0000 2400;.  local access number and baud rate
A="Your Town -Class B";.   type of Telenet access for msg later
key i="PCP_USERID";.       key for entering user ID
key w="PCP_PW";.           key for entering password

%m=100;.                   default menu (100 -> display it)
%t=5;.                     default attempts to reach city
%s=1;.                     default attempts to reach BBS
%b=2400;.                  default baud rate value for outdial
B="24"                     default baud rate as a string
%n=2;.                     automatic stepdown mode
key 0="read pcpmenu/r";    key for reinvoking script

.. ------------------------------------------------------------
..
..      Initialization
..
.. ------------------------------------------------------------

.. Initialize data from command line.  There are three optional
.. parameters.  The first is a menu selection number to call
.. immediately.  The second is the number of attempts that
.. should be made to connect to the destination city.  The third
.. is the number of times to attempt to connect to the local
.. number in that city.  In all cases, we check to make sure that
.. we have an acceptable value.

.. menu selection (variable %m)

%d={1:0);.             read parameter 1 with default value of 0
if %d<1 GOTO BAD1;.    ignore if illegal value
if %d>100 GOTO BAD1
%m=%d;.                if value in range 1..100, use it
LABEL BAD1

.. similar code for other parameters omitted.

.. Initialize various variables and MEX parameters

%a=0;.                 no area code requested
.. etc.
D=" ";.                no PCP outdial (city) code
.. etc.

stat filter on         turn filter on
stat trigger "";.      do sendouts immediately
stat sodelay on;.      ..but at a slow rate
stat reply 0;.         do not wait for response to a sendout
stat case on;.         ignore case
.. etc.

.. ------------------------------------------------------------
..
..      Initialize the Modem
..
.. ------------------------------------------------------------

.. Make sure the modem is connected and responding.

sendout "AT/r"
wait string 2 "OK" "0";.      allow verbose or terse responses
if value=0 A="No response from local modem";GOTO ABORT
.. etc.

.. ------------------------------------------------------------
..
..      Establish Connection to Local Access to Telenet
..
.. ------------------------------------------------------------

screen on;.                    tell user what we are doing
say "/n/nDialing Telenet (",A,") . . . ";.
screen off

.. place call to Telenet

dial pcp
if value=0 A="No connection to Telenet:";GOTO ABORT
screen on
say "CONNECTED/n"

.. initialize PCP session

%z=2;.                         max tries
LABEL LOGIN
if %z<1 A="Telenet not responding";GOTO ABORT
%z=%z-1
screen on;say "  sync... ";screen off
sendout "@";sleep 1;sendout "D/r"
wait string 1 "TERMINAL"
if value=0 sleep 1;goto LOGIN
screen on;say "terminal ID... ";screen off
sendout "D1/r"

read PCPMENU;.                 chain to PCPMENU script

.. ------------------------------------------------------------
..
..      Subroutines
..
.. ------------------------------------------------------------

LABEL ABORT
screen on
say "/n/n",A,"; session aborted./n/n"
dsc
```

The menu of PCP cities is drawn by calling the CITYNAME routine from inside a loop. This is much slower than drawing the menu directly. I put the code for the CITYNAME subroutine at the beginning of the script, since I think it executes a little faster from there.

I wrote the script this way for two reasons. First, it illustrates some interesting techniques, such as iterated loops and computed coordinates for the "@" command. Second, it keeps information about the city names in one place. If they are kept in more than one place, then when changes are made in the future they might not be made everywhere. As it is, this danger exists in several places in these scripts. For example, the menu of RASs is drawn in PCPMENU, but the data for each RAS are stored in PCPDATA. When changes are made, the user must be sure to keep the information synchronized.

Script PCPCALL.MEX

Now we come to the most complex module in the script (Listing 5). This one has to perform a lot of housekeeping and tricky operations. It has to know if we are already connected to a city and if so which one. It then has to decide how to go about connecting to the requested city. This may require disconnecting from the

(Text continued on page 15)

Table 1: A list showing how the string and numerical variables are used in the scripts.

```
A       error message string, temporary string
B       baud rate
C       temporary city code in PCPMAN
D       PCP outdial city code
E       RAS phone number
F       RAS system name, scratch sometimes

%a      area code of city currently connected to (or 0 if none)
%b      maximum baud rate
%c      current city selection (see PCPMAN.MEX menu)
%d      temporary variable
%e      max number of tries
%f      main area code
%g      alternate area code (or 0 if none)
%h      flag for "1" prefix requirement (0 if not required)
%i
%j
%k
%l
%m      menu selection
%n      stepdown mode ( 0 = manual, >0 = automatic )
%o      flag used to indicate current stepdown status (0 = no
        ..stepdown yet)
%p      number of city actually connected to
%q
%r      count of number of rings detected
%s      default tries to connect to RAS
%t      default tries to connect to city
%u      flag to show intial attempt to connect to a city or RAS
%v      flag to show Vadic status ( >0 if modem in Vadic mode )
%w      cursor row
%x      loop index
%y      cursor column
%z      dummy counter
```

Listing 2: Script PCPMENU.MEX

```
.. PCPMENU.MEX  (04/15/90)

if %m<100 GOTO PROCESS;.  if selection already made, process it

.. ------------------------------------------------------------
..
..      Main Re-entry Point and Display of BBS Menu
..
.. ------------------------------------------------------------

LABEL DRAWSCREEN

screen on
cls
box 1 1 20 79
@ 2 32 say "PC-PURSUIT MENU"

@  6 3 say "1. Al Hawley (ZN02)"
.. etc.
@ 14 3 say "9. Vanhorn (ZN66)"

@  6 27 say "10. Roger Warren (ZN09)"
.. etc.
@ 14 27 say "18. ....................."

@  6 52 say "19. Terry Pinto"
@ 11 52 say "24. ....................."
.. etc.
@ 12 52 say "25. ....................."
@ 14 52 say "27. WLA PCBoard"

@ 16 12 say "  0. RESET"
@ 17 12 say " 99. QUIT"
@ 18 12 say "100. MANUAL ENTRY MENU"

.. The following baud-rate-selection choices are allowed only if
.. not presently connected to a city.

if %a=0
  @ 16 45 say "200. 2400 BAUD AUTO"
  @ 17 45 say "201. 1200 BAUD FIXED"
  @ 18 45 say "202. 2400 BAUD FIXED"
endif

.. ------------------------------------------------------------
..
..      Baud Rate / City Connection Status Display
..
.. ------------------------------------------------------------

LABEL DRAWBAUD

if %a=0
  if %n>0
    @ 4 20 say "  Set to max ",%b," bps (auto stepdown)      "
  else
    @ 4 20 say "            Set to ",%b," bps          "
  endif
else
  @ 4 20 say "Connected to Citycode ",D," at ",B,"00 bps"
endif

.. ------------------------------------------------------------
..
..      Get / Process Selection
..
.. ------------------------------------------------------------

LABEL GETSEL

@ 22 12 say "Enter Selection:       "
@ 22 29
input;%m=value

LABEL PROCESS

.. ---------- special selections

if %m=0   GOTO RESET
if %m=99  A="User termination";GOTO ABORT
if %m=100 READ PCPMAN;.      manual entry of city and phone #

.. if already connected to a city, do not allow baud rate changes
.. by resetting the choice to 9999

if %p<>0
  if %m>=200 %m=9999
endif

.. handle baud mode selections

if %m=200 %n=2;%b=2400;B="24";GOTO DRAWBAUD
if %m=201 %n=0;%b=1200;B="12";GOTO DRAWBAUD
if %m=202 %n=0;%b=2400:B="24";GOTO DRAWBAUD

if %m>27 GOTO BADSELECT
if %m<1  GOTO BADSELECT

.. ---------- calling selections

.. reject unassigned menu choices

if %m=12 GOTO BADSELECT
.. etc.
if %m=25 GOTO BADSELECT

READ PCPDATA;.              chain to data fetch routine

LABEL BADSELECT

bell 1
GOTO GETSEL

.. significant part of script omitted

.. ------------------------------------------------------------
..
..      Subroutines
..
.. ------------------------------------------------------------

LABEL ABORT

screen on
cls
@ 5 0 say A,"; session ended."

%z=3;.                      max tries

LABEL ABORT1
%z=%z-1
if %z<1 dsc;GOTO ABORT2;.   if soft disconnect fails, hang up
screen on
sendout "/r@/r";.           send disconnect commands to Telenet
wait string 1 "@"
if value<>1 GOTO ABORT1
sendout "hangup/r"
wait string 10 "NO CARRIER"
if value <>1 GOTO ABORT1

LABEL ABORT2
sendout "AT\N1/r"
say "/n/nModem MNP mode turned off/n"
cpm
```

Listing 3: Script PCPDATA.MEX

```
.. PCPDATA.MEX  (04/14/90))
..
.. This routine contains all the data for the systems in the
.. menu.  Each entry sets the following variables:
..
..      B               actual baud rate code ("12" or "24")
..      D               outdial code
..      E               phone number
..      F               name of system called
..      %a              area code
..      %c              city number (see PCPMAN menu)
..
.. In addition, function keys are set up (where the defaults are
.. not correct) with the login names and/or passwords.

key 1="Your Name/r";.       default key definitions
key 2="Your-Usual-PW/r"

if %b=1200 B="12"
if %b=2400 B="24"
.. dispatch table

if %m=1  GOTO BBS01
.. etc.
if %m=27 GOTO BBS27

.. ------------------------------------------------------------
..
..      Data on Remote Access Systems from Menu
..
.. ------------------------------------------------------------

LABEL BBS01
F="Al Hawley (ZN2)"
D="CALAN"
%c=13
%a=213
E="670-9465"
key 1="###CCCCCCC/r";.      login id/password combination
READ PCPCALL;.              chain to place the call

.. entries for other systems omitted
```

Listing 4: Script PCPMAN.MEX

```
.. PCPMAN.MEX  (04/14/90)

.. This subroutine script handles the manual entry of data for a
.. destination RAS.

.. See if there is usable call data already defined.  If so, the
.. user will be given the option of using it (in case this is a
.. second attempt to make the same manual call).  Otherwise, we
.. will skip that menu and go directly to the data input menus.

GOTO START

.. ---------------------------------------------------------------
..
..      Subroutines (placed at beginning for speed)
..
.. ---------------------------------------------------------------

.. This subroutine is passed the city number in scratch variable
.. %d and returns the name of the city in string variable A.

PROC CITYNAME
if %d=1 A="Atlanta, GA";ENDP
.. etc.
if %d=34 A="Washington, DC";ENDP
A="Unknown City"
ENDP

.. ---------------------------------------------------------------

.. This subroutine takes the city number in %d and sets the PCP
.. city code into string variable C, the main area code into
.. variable %f, and any additional area code into %g.  Variable
.. %h is set to 1 if a "1" prefix is required.

PROC PCPCODE

.. initialize

%f=0;.          no main area code
%g=0;.          no alternate area code
%h=0;.          no "1" prefix needed
C=" ";.         no city code

if %d=1  C="GAATL";%f=404;ENDP
if %d=2  C="MABOS";%f=617;ENDP
if %d=3  C="ILCHI";%f=312;%g=815;%h=1;F="815";ENDP
.. etc.
if %d=29 C="CASJO";%f=408;%g=415;F="415";ENDP
.. etc.
if %d=34 C="DCWAS";%f=202

ENDP

.. ---------------------------------------------------------------
..
..      Main Code
..
.. ---------------------------------------------------------------

LABEL START

C=D;.           set temporary city code to current value
if %c=0 GOTO SETCITY
comp C " ";if value=1 GOTO SETCITY
comp E " ";if value=1 GOTO SETCITY

screen on

.. ---------------------------------------------------------------
..
..      Ask About Entering New RAS Data
..
.. ---------------------------------------------------------------

cls

%d=%c;.         convert city code into city name in A
GOSUB CITYNAME

box 1 5 15 50
@  3 19 say "Manual Call Entry"
@  5 10 say "1. use current data"
@  6 15 say "System Name:     ",F
@  7 15 say "City:            ",A
@  8 15 say "PCP City Code:   ",C
@  9 15 say "Data Rate:       ",%b
if %n>0 say " AUTO"
@ 10 15 say "Area Code:       ",%a
@ 11 15 say "Phone Number:    ",E
@ 13 10 say "2. enter new RAS data"

@ 17 1 say "Enter selection (or 0 to return to main menu):"

LABEL ASKMODE

@ 17 48 say "       "
@ 17 48
input

if value=3 %m=100;READ PCPMENU
if value=1 GOTO MAKECALL
if value<>2 bell 1;GOTO ASKMODE

.. ---------------------------------------------------------------
..
..      Get PCP Outdial City Code
..
.. ---------------------------------------------------------------

LABEL SETCITY

cls
box 1 1 18 79

if %p=0
  @ 3 28 say "PC-Pursuit City Choices"
else
  %d=%p
  GOSUB CITYNAME;.          get current city name into A
  @ 3 10 say "PC-Pursuit Cities (Currently Connected to ",A,")"
endif

%x=1;.                          starting city number
LABEL COLUMN
%d=%x
GOSUB CITYNAME
%y=4;%w=%x+4
if %x<10 %y=5
if %x>12 %y=28;%w=%x-8
if %x>24 %y=52;%w=%x-20
@ %w %y say %x,". ",A
%x=%x+1
if %x<35 GOTO COLUMN

@ 20 5 say "City code selection (or 0 to return to main menu):"

LABEL ASKCITY

@ 20 56 say "       "
@ 20 56
input;%c=value

if %c=0 %m=100;READ PCPMENU;.   return to main menu
%d=%c;GOSUB PCPCODE;.           set PCP code and area code data
if %c=%p GOTO SETPHONE;.  if already connected, skip baud query

.. ---------------------------------------------------------------
..
..      Get baud rate to use if a new city is specified.
..
.. ---------------------------------------------------------------

LABEL SETBPS

cls
box 5 5 13 54
@  7 10 say "Data rate to use for new city (",C,"):"
@  9 15 say "1.  2400 AUTO"
@ 10 15 say "2.  1200 FIXED"
@ 11 15 say "3.  2400 FIXED"
@ 15 1 say "Enter selection (or 0 to return to main menu):"

LABEL ASKBPS
@ 15 48 say "       "
@ 15 48
input

if value=1 %b=2400;B="24";%n=2;GOTO SETPHONE
if value=2 %b=1200;B="12";%n=0;GOTO SETPHONE
if value=3 %b=2400;B="24";%n=0;GOTO SETPHONE

.. if bad answer, ring bell and get another answer

bell 1
GOTO ASKBPS

.. ---------------------------------------------------------------

LABEL SETPHONE

cls

if %g=0 %a=%f;GOTO SETNUMB

box 5 5 12 50
@  7 10 say "Area Codes for PCP City Code ",C
@  9 15 say "1. main area code:      ",%f
@ 10 15 say "2. alternate area code: ",%g

@ 14 10 say "Area code selection: "

LABEL ASKAREA

@ 14 32 say "       "
@ 14 32
input

if value=1 %a=%f;%h=0;GOTO SETNUMB
if value=2 %a=%g;GOTO SETNUMB

.. if invalid response, ring bell and ask again

bell 1
GOTO ASKAREA
```

(Listing 4 continued on next page)

(Listing 4 continued from previos page)

```
.. ------------------------------------------------------------

LABEL SETNUMB

cls
say "/n/nArea code is ",%a

say "/n/nEnter phone number (###-####): "
accept E,8

if %a=%g E="{F}-{E}";.  prefix the area code to the number
if %h>0  E="1-{E}";.    if required, prefix "1-" to number

say "/nEnter system name (optional): "
accept F

%d=%c;GOSUB CITYNAME

label MAKECALL
say "/n/nReady to place the following call:/n"
say "/n   System Name:    ",F
say "/n   City:           ",A
say "/n   PCP City Code:  ",C
say "/n   Data Rate:      ",%b
if %n>0 say " AUTO"
say "/n   Area Code:      ",%a
say "/n   Phone Number:   ",E

say "/n/nData Correct (Y//n)? "
accept A,1

comp ".{A}" ".";if value=1 GOTO MAKECALL1
comp A "Y";if value=1 GOTO MAKECALL1
GOTO SETCITY;.                          start over again

LABEL MAKECALL1

D=C;.                           set PCP city code
READ PCPCALL;.                  chain to PCPCALL script
```


SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($70)
  - NZCOM: Z-System for CP/M-2.2 computers ($70)
  - ZCPR34 Source Code: if you need to customize ($50)
- Plu*Perfect Systems
  - Backgrounder ii: CP/M-2.2 multitasker ($75)
  - ZDOS: date-stamping DOS ($75, $60 for ZRDOS owners)
  - DosDisk: MS-DOS disk-format emulator, supports subdirectories and date stamps ($30 - $45 depending on version)
- BDS C — Including Special Z-System Version ($90)
- Turbo Pascal — with New Loose-Leaf Manual ($60)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Z80 Assemblers using Zilog (Z80ASM), Hitachi (SLR180), or Intel (SLRMAC) Mnemonics
  - Linker: SLRNK
  - TPA-Based ($50 each) or Virtual-Memory (Special: $160 each)
- ZMAC — Al Hawley's Z-System Macro Assembler with Linker ($50)
- NightOwl (Advanced Telecommunications)
  - MEX-Plus: automated modem operation with scripts ($60)
  - MEX-Pack: remote operation, terminal emulation ($100)

Next-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $3 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (pw=DDT) (MABOS on PC-Pursuit)

Listing 5: PCPCALL.MEX

```
.. PCPCALL.MEX  (04/16/90)

.. This script performs the steps necessary to connect to the
.. designated city and remote system.

%m=100;.        clear menu selection
%u=0;.          flag to show initial run thru script

.. *** section omitted here ***

.. Branch depending on whether we are presently connected to no
.. city, to the requested city, or to a different city.

if %p=0  GOTO NEWCITY;.     no city connected presently
if %c=%p GOTO LOCAL;.       already connected to desired city

.. ------------------------------------------------------------
..
..      Disconnect from wrong city
..
.. ------------------------------------------------------------

%d=%p;GOSUB CITYNAME;.    get name of currently connected city
say "/n/nDisconnecting from ",A," . . . "

%z=3;.                    max number of tries

LABEL DROPCITY

screen off
sendout "/r@/r";.         try to return to PCP command mode
wait string 2 "@";.       we should get '@' prompt
if value>0 GOTO DROP1;.   if we do, continue below
%z=%z-1;.                 else decrement count
.. abort if count expired
if %z<1 A="Cannot quit current city";GOTO ABORT
GOTO DROPCITY;.           try again

LABEL DROP1
screen off
sendout "D/r";.           tell PCP to disconnect from city
%p=0;.                    show no city connected
%v=0;.                    show not in Vadic mode
screen on
say "OK";.                tell user that it worked

.. ------------------------------------------------------------
..
..      Connect to new city
..
.. ------------------------------------------------------------

LABEL NEWCITY

screen on

.. We do different things depending on whether or not we have
.. exhausted the first set of attempts.

if %u=0
  say "/n";.              end any line of screen output
  %u=1;.                  show no longer first time
  %e=%t;.                 number of tries into variable 'e'
else
  A=" city code"
  gosub GETD;.            gets number of times to try
  say "/nCalling ",F,"/n";.tell user whom we are trying to reach
endif

%d=%c;GOSUB CITYNAME
say "/nDialing city code ",D," (",A,")";.report the city code we
                                              are dialing

%z=1;.                    count of tries
%o=1;.                    indicate no auto stepdown yet

LABEL CITYCODE

if %z>%d goto MORETRIES
sleep 1
screen on;say "/n  try #",%z," of ",%e," at ",B,"00bps..."
                                                ;screen off
%z=%z+1
sendout "C D//";sendout D;.          city code
sendout "//";sendout B;.             baud code
sendout ",";sendout "PCP_USERID";.   user id
sendout ",";sendout "PCP_PW";.       user password
sendout "/r"

wait string 4 "CONNECTED" "BUSY" "FAIL"

.. If we connected, set %p to show new city connected and %u to
.. indicate the first pass at connecting to the specified RAS.

if value=1 %u=0;%p=%c;GOTO LOCAL

screen on
if value=3 say "Failed Call";goto MORETR1
if value=2 say "Busy"
```

(Listing 5 continued on next page)

(Listing 5 continued from previous page)

```
if value=0 say "No Response"
goto CITYCODE

LABEL MORETRIES
if %n=0 goto MORETR1;.    no auto step down
if %o=0 goto MORETR1;.    already stepped down

%o=0;.                    show now stepped down
if %b=2400 B="12";%z=1;GOTO CITYCODE;.  reset trial count %z

LABEL MORETR1

A=" city code"
GOSUB ASKMORE
if value=1 GOTO NEWCITY;. try again to reach new city

READ PCPMENU;.            else chain back to main menu

.. ------------------------------------------------------------
..
..      In right city; try connecting to specified system
..
.. ------------------------------------------------------------

LABEL LOCAL
screen on
%d=%p;GOSUB CITYNAME
say "/n/nConnected to city code ",D," (",A,")"

.. initialize destination modem to make sure it is still alive

if %v=2;.                 if in Vadic mode, reinitialize modem
  sendout "I/r";.         exit from it
  %v=0;.                  show initialized modem
  sleep 1
endif

LABEL REINIT
%z=3;.                    max attempts to initialize modem
LABEL REINIT1
if %z<1 A="Remote modem failure";goto ABORT
%z=%z-1
screen on;say "/n  initialize remote modem... ";screen off
sleep 1
sendout "ATZ/r"
wait string 3 "OK"
if value=0 goto REINIT1

.. Call destination system

LABEL LOCALAGAIN

.. *** section omitted ***

LABEL DIAL

sendout "^E/r";.          enter Vadic mode
wait string 3 "*"
screen on
if value=0 say "/nVadic Mode Failed/n";goto REINIT

say "/nRemote modem now in Vadic Mode"
%v=2;.                    show Vadic mode

LABEL DIAL1

%r=0;.                    initialize ring count
screen on;say "/n/nTry #",%z," of ",%e," (at ",B,"00 bps)/n"
sendout "D";sendout E;sendout "/r"

LABEL DIAL2
screen on
wait string 25 "ANSWER" "BUSY" "DIAL TONE" "RINGING..."
if value=1  goto SUCCESS
if value=2  goto BUSY
if value=3  goto DIALTONE
if value<>4 goto BADCODE

.. *** many routines omitted ***

LABEL SUCCESS
screen on
say "^G/n/nCONNECTED TO HOST SYSTEM/n/n"
READ PCPCONN;.            chain to short script to free memory


.. ---------- Subroutines

.. Ask user for the number of times to connect and put answer is
.. %e.  If the answer is less than 1, then use a value of 1.

PROC GETD
say "/nTry how many times to connect to",A,"? "
input
%e=value
if %e<1 %e=1
ENDP

.. Ask if user wants to continue trying to connect.  Return with
.. the answer in VALUE, 1 if YES, 0 if NO.

PROC ASKMORE
LABEL ASKAGAIN
screen on
say "/n/nTry",A," some more (Y//N)? "
bell 1
accept C,1
screen off
comp C "Y"
if value=1 ENDP
comp C "N"
if value=1 value=0;ENDP
bell 1
goto ASKAGAIN

.. *** remaining routines omitted ***
```

Listing 6: Script PCPCONN.MEX

```
.. PCPCONN.MEX  (02/08/90)

.. This short script is run once the destination system has been
.. reached.

screen on
cls
say "Connected to ",F," at ",B,"00 bps/n/n"
t;.                              enter terminal mode

LABEL LOOP
say "/nEnter a single MEX command (or M for menu): "
accept A
comp A "M"
if value=1 READ PCPMENU
{A}
GOTO LOOP
```

current city and then connecting to the new city.

The script has to allow for things not always going right, at least not the first time. You should particularly note the pains it takes to reset and test the PCP outdial modem and to put it into Racal-Vadic mode.

If the outdial modem is busy or if the RAS is busy, the script will ask the user whether to make additional attempts and if so how many. The script is very careful to keep the user informed of exactly what is going on.

If all works out, we eventually end up connected to the remote system. PCPCALL then chains to PCPCONN (Listing 6), which drops one into terminal mode with the function keys programmed to ease logging in. It would not be hard to have the script first call a subroutine script to perform the login operation automatically. The easiest way would be to store a number in a numerical variable as a flag and the name of the login script file in a string variable. Unfortunately, it's not clear that we have a free string variable to use. One possibility would be to use the system name in variable F. One could run it using the command "DO {F}".

Conclusion

I hope this extended example will give you a better idea of how and to what extent MEX script commands can be used to automate telecommunications tasks. Please let me know if you have some ideas to improve these scripts. ●

The Z-System and Turbo Pascal

by Joe Wright, Alpha Systems

"Real Programmers Don't Eat Quiche" really started something. Real Programmers don't write applications, only tools. Real Programmers don't write in High-Level languages, only in Assembler. And so on...

These statements become popular cliches only because they are largely true. Authors of High-Level languages strive for generality and avoid system-specific constructs or references. Authors of system-level programs find High-Level languages insufficient to the task and almost always choose Assembler.

I suppose I am primarily a system-level or "Real" programmer and have never had much use for High-Level stuff (I mean, can you really write a BIOS in Fortran?). I have written a few 'application' programs in dBASE II (a High-Level language) but have spent most of my time writing 'tools' in Assembler.

As some of you already know, I have had an abiding interest in ZCPR3 and the Z-System for some years now. I am the author of NZCOM and its predecessors dating back to 1985. My company, Alpha Systems, began distributing Borland's Turbo Pascal 3.0 (CP/M and PC-DOS versions) earlier this year. In order to handle Tech Support phone calls, I bent myself to the awesome task of learning a new High-Level language.

Well, its not really that hard. Pascal was designed as a means to teach programming (to un-real programmers?) and does the job well. I'm learning.

The Z-System derives much of its power and flexibility from the use of complex data structures such as Z3ENV and Z3MSG segments. Attempts to address these structures in the context of a High-Level language have not borne fruit, until now. Please let me digress for a moment.

A Z3 utility (.COM file) has a special 'header' associated with it which includes a 'pointer' to Z3ENV, the Environment Descriptor. Knowing the address and the structure of Z3ENV, the Z3 utility can have complete access to the entire Z-System, especially through the use of SYSLIB, Z3LIB and VLIB subroutine libraries. Most of us know this already. Let's continue with something I just found out.

Turbo Pascal has a Record construct allowing the definition of complex data structures as variables in Pascal terms. This means we can define Z3ENV, Z3MSG, Z3NDIR, Z3CL and other Z-System structures as Pascal records of variables. If we only knew where they were.

Further, Pascal has a Pointer construct to point to a record of variables. If we only knew where the pointer was.

OK, you guessed it. We patch TURBO.COM so that it looks like a Z3 utility and can be 'installed' by the command processor which puts the Z3ENV address at 109H in the .COM file. In Pascal, we define the contents of 109H as a Pointer variable to a Record of variables, the Environment Descriptor. From now on, we can access the entire Z-System as Pascal variables.

I apologize if this seems a bit obtuse to the casual reader. In effect, the High-Level Application programmer in Pascal can now have complete access to Z-System and the "Real" system-level programmer, who can't even spell quiche, is no longer restricted to Assembler. Many otherwise complex Z-System "tools" can be written quickly and easily in Turbo Pascal.

To demonstrate this, I have written NZ-TOOL.BOX as shown in Listing 1 for Turbo Pascal. It contains the Z-System structures defined as Records and a few SYSLIB and Z3LIB functions in Pascal.

I offer this tool.box and the simple programs which follow with abject humility and apology to the veteran Pascal programmer. I am told that "It doesn't even look like Pascal" by more than one. "It looks like Assembly Language", they say.

Well, what do you expect from Joe? I have been programming in Assembler for six years and in Pascal for six weeks. Style aside, it works. Please dress it up for me. Thanks.

The CPY and PD programs shown in Listing 2 are perhaps too simple to be useful Z3 utilities. They are here to demonstrate the use of the NZ-TOOL.BOX routines.

Well, I'm out of breath (and out of paper) and will get off my soapbox for now. I hope that I have tweaked more than a little interest in both Z-System and Turbo Pascal. In the case that I have succeeded, you can get either or both from:

Alpha Systems Corporation
711 Chatsworth Place
San Jose, CA 95128
(408) 297-5594

Listing 1

```
(* NZ-TOOL.BOX 1.0 for Turbo Pascal 3.0
   Copyright (C) 1989 Alpha Systems

   Author:  Joe Wright
   Date:    20 Sept 89
   Version: 1.0

   Invoke TURBO.COM with a Z-System alias, TP.COM, as follows..

     1 --> GET 100 TURBO.COM;
     2 --> POKE 103 5A 33 45 4E 56 01 00 00;
     3 --> GO

   This puts a Z3ENV Type 1 header at the beginning of TURBO.COM
   so that GO will 'install' the Environment address at 109H.
   COM files created by TURBO.COM will also contain this header
   and therefore be Z3 utilities.

   This version of the TOOL.BOX describes the entire Z-System in
   terms of Turbo Pascal Records.  The Record structure and Turbo
   Pascal Pointers will allow the veteran Pascal programmer complete
   access to the Z-System Environment and by implication, a
description
   of its entire structure.

   The Z-System assembly language programmer is presented with Pascal
   functions and procedures with familiar SYSLIB and Z3LIB names so
   that he can 'call' his favorite subroutines within Turbo Pascal
   without re-inventing the wheel.

   Programming style seems to vary according to the square of
   of programmers (or square programmers?) and no one should feel
   limited by the examples in this TOOL.BOX.  Think of it as a
   demonstration.  In the end, do it your way.

   Load the Tool Box into you program with the Turbo Pascal
   ($I NZ-TOOL.BOX) insertion directive.

   Turbo Pascal is not upper/lower case sensitive.  We tend to use
   mixed case for readability.  In order to distinguish among Turbo
   Pascal Standard declarations and our own, I try to follow the
   convention that..
```

(Listing 1 continued on next page)

(Listing 1 continued from previous page)

```
    GoToBed   is a Turbo Pascal defined Word or Identifier
    GOTOBED   is the User Definition
    gotobed   is the User Declaration

  Mixed case is Pascal, UPPERCASE is our own definition and
  lowercase declares our definitions.

*)

{ User-Defined Global Types }

Type
  STR8    = String[8];              { Dir Name or Password          }
  STR21   = String[21];             { NDIRNAME:FILENAME.TYP         }

  ARR8    = Array[1..8] of Char;    { Name or Password array        }
  ARR3    = Array[1..3] of Char;    { File type array               }
  ARR5    = Array[1..5] of Char;    { Five-char ID like Z3ENV       }

  SECTOR  = Array[0..127] of Byte;  { Standard CP/M Unit Record     }

  FCBPTR  = ^FCBREC;
  FCBREC  =
    Record
      DRIV  : Byte;
      NAME  : arr8;
      TYP   : arr3;
      EXT   : Byte;
      S1    : Byte;
      S2    : Byte;
      RC    : Byte;
      ALLOC : Array[0..15] of Byte;
      CR    : Byte;
      RREC  : Integer;
      RERR  : Byte
    End;

{ This describes the Z3MSG structure. }

  MSGPTR = ^MSGREC;
  MSGREC =
    Record
      ERFLG   : Byte;
      IFLEV   : Byte;
      IFSTS   : Byte;
      CMDST   : Byte;
      ERADR   : Integer;
      PRGER   : Byte;
      ZEXMSG  : Byte;
      ZEXRUN  : Byte;
      ZEXNXT  : Integer;
      ZEX1ST  : Integer;
      SHCTL   : Byte;
      SCRAT   : Integer;
      ERCMD   : Array[1..32] of Char;
      REGIS   : Array[0..31] of Byte;
    End;

{ This describes the Z3CL structure for 'standard' Z-Systems. }

  MCLPTR = ^MCLREC;
  MCLREC =
    Record
      NXTCHR  : Integer;
      MCLMAX  : Byte;
      MCL     : String[203];
    End;

  MEMORY = Array[0..$7FFE] of Integer;

  PATPTR = ^PATREC;
  PATREC =
    Record
      PATH : memory;
    End;

  NDRPTR = ^NDRREC;
  NDRREC =
    Record
      DU : Integer;
      NAME : arr8;
      PASS : arr8;
    End;

{ The following Record Structures define the Z3 Environment of
  any NZ-System.  }

  ENVPTR = ^ENVREC;
  ENVREC =
    Record
      ENV     : Byte;
      CBIOS   : Integer;
      Z3ID    : Array[1..5] of Char;
      ENVTYP  : Byte;
      EXPATH  : patptr;
      EXPATHS : Byte;
      RCP     : Integer;
      RCPS    : Byte;
      IOP     : Integer;
      IOPS    : Byte;
      FCP     : Integer;
      FCPS    : Byte;
      Z3NDIR  : ndrptr;
      Z3NDIRS : Byte;
      Z3CL    : mclptr;
      Z3CLS   : Byte;
      Z3ENV   : Integer;
      Z3ENVS  : Byte;
      SHSTK   : Integer;
      SHSTKS  : Byte;
      SHSIZE  : Byte;
      Z3MSG   : msgptr;
      EXTFCB  : fcbptr;
      EXTSTK  : Integer;
      QUIET   : Byte;
      Z3WHL   : ^Byte;
      SPEED   : Byte;
      MAXDSK  : Byte;
      MAXUSR  : Byte;
      DUOK    : Byte;
      CRT     : Byte;
      PRT     : Byte;
      COLS    : Byte;
      ROWS    : Byte;
      LINS    : Byte;
      DRVEC   : Integer;
      SPAR1   : Byte;
      PCOL    : Byte;
      PROW    : Byte;
      PLIN    : Byte;
      FORM    : Byte;
      SPAR2   : Byte;
      SPAR3   : Byte;
      SPAR4   : Byte;
      SPAR5   : Byte;
      CCP     : Integer;
      CCPS    : Byte;
      DOS     : Integer;
      DOSS    : Byte;
      BIO     : Integer;
      SHVAR   : Array[1..11] of Char;
      FILE1   : Array[1..11] of Char;
      FILE2   : Array[1..11] of Char;
      FILE3   : Array[1..11] of Char;
      FILE4   : Array[1..11] of Char;
      PUBLIC  : Integer;
    End;

{ Global Variables }

{ These are the Absolute (External) variable assignments which
  give the Turbo Pascal program access to everything we know. }

Var
  WBOOTV  : Integer     Absolute $0001;
  IOBYTE  : Byte        Absolute $0003;
  CDISK   : Byte        Absolute $0004;
  BDOSV   : Integer     Absolute $0006;
  FCB1    : fcbrec      Absolute $005C;
  FCB2    : fcbrec      Absolute $006C;
  TBUFF   : String[126] Absolute $0080;
  DBUFF   : sector      Absolute $0080;
  Z3EADR  : envptr      Absolute $0109;

{ These Global Variables are used by the new Functions and Procedures
  and the Main program to pass parameters among themselves. }

  SPEC1   : str21;      {For DIRECTRY:FILENAME.TYP}
  SPEC2   : str21;      {For DIRECTRY:FILENAME.TYP}
  DIRS    : str8;       {For D:, U:, DU: or DIR:}
  NAME    : str8;       {Directory Name}
  PASS    : str8;       {Directory Password}

  SOURCE  : File;       {File being Read}
  DESTIN  : File;       {File being Written}
  CURDU   : Integer;    {Current (default) Drive/User}
  SRCDU   : Integer;    {Source D/U}
  DSTDU   : Integer;    {Destination D/U}


{ From here on, we will collect various User-Defined functions
  and procedures which emulate SYSLIB and Z3LIB subroutines of
  the Z-System.  To the extent that they may 'call' each other,
  they are arranged here such they are Declared before they are
  called, SYSLIB, Z3LIB, VLIB, in that order.  Turbo Pascal is
  itself rich enough to provide most of the functions we need. }

Procedure CAPSTR(Var S:str21);
Var I : Integer;
Begin
  for i := 1 to Length(s) do s[i] := UpCase(s[i])
End;

Function PHEX(N,B:Integer):str8;
Var H : str8; I, C : Byte;
Begin
  h[0] := #0; { Clear the string }
  For i := b Downto 1 Do
  Begin
    c := n and 15; n := n shr 4;
    If c < 10 Then c := c+48 Else c := c+55;
    Insert(Chr(c),h,1)
  End;
  phex := h
End;
```

(Listing 1 continued on next page)

(Listing 1 continued from previous page)

```
Function RETUD:Integer;
Begin
  retud := 256 * (Bdos(25)+1) + Bdos(32,$FF)
End;

Procedure LOGUD(DU:Integer);
Begin
  Bdos(14,Hi(DU)-1); Bdos(32,Lo(DU))
End;

Procedure PDU(du:Integer);
Begin
  Write(Chr(Hi(du)+64),Lo(du),':')
End;

Function GETNDR:Integer;
Begin
  getndr := Ord(z3eadr^.z3ndir)
End;

Function DIRSCAN(S:str8):Integer;
Var NDIR : ndrptr;
    D, I : Integer;
Begin
  d := 0;
  ndir := Ptr(getndr);
  if ndir^.du <> 0 then
    Repeat
      name[0] := #8;
      for i := 1 to 8 do name[i] := ndir^.name[i];
      i := Pos(' ',name);
      if i <> 0 then name[0] := Chr(i-1);
      if s = name then d := Swap(ndir^.du) else d := 0;
      ndir := Ptr(Ord(ndir)+SizeOf(ndrrec));
    Until (d <> 0) or (Lo(ndir^.du) = 0);
  dirscan := d;
End;

Function DUSCAN(S:str8):Integer;
Const d = 'ABCDEFGHIJKLMNOP';
Var du, usr, cod : Integer;
Begin
  du := curdu;
  if Length(s) <> 0 then
    begin
      if Pos(s[1],d) <> 0 then
        begin
          du := Pos(s[1],d)*256 + (du and 255);
          Delete(s,1,1)
        end;
      if Length(s) <> 0 then
        begin
          Val(s,usr,cod);
          if cod <> 0 then
            du := curdu else du := (du and -256) + (usr and 31)
        end;
    end;
  duscan := du;
End;

Function DNSCAN:Integer;
Var du : Integer;
Begin
  du := dirscan(dirs);
  if du = 0 then du := duscan(dirs);
  dnscan := du
End;

Function NAMSTR(Nam:arr8):str8;
Var I : Integer; Str : str8;
Begin
  str[0] := #8;
  for i := 1 to 8 do str[i] := nam[i];
  i := Pos(' ',str);
  if i <> 0 then
    str[0] := Chr(i-1);
  namstr := str
End;

Function DUTDIR(du:Integer):Boolean;
Var ndir : ndrptr;
    i    : Integer;
Begin
  name[0] := #0;
  pass[0] := #0;
  ndir    := Ptr(getndr);
  Repeat
    if du = Swap(ndir^.du) then
      begin
        name := namstr(ndir^.name);
        pass := namstr(ndir^.pass);
      end;
    ndir := Ptr(Ord(ndir)+SizeOf(ndrrec));
  Until (Length(name) <> 0) or (Lo(ndir^.du) = 0);
  dutdir := Length(name) <> 0
End;

Function GETWHL:Boolean;
Begin
  if z3eadr^.z3whl^ = 0 then
    getwhl := false
  else
    getwhl := true
End;

Procedure SETWHL(B:Byte);
Begin
  z3eadr^.z3whl^ := b
End;

Function GETDUOK:Boolean;
Begin
  if z3eadr^.duok = 0 then
    getduok := false
  else
    getduok := true
End;

Procedure PARSE(Var S:str21);
Var P : Integer;
Begin
  dirs[0] := #0;  { Clear the string }
  p := Pos(':',s);
  if p <> 0 then
    begin
      dirs := Copy(s,1,p-1);
      Delete(s,1,p)
    end;
End;

{ End of NZ-TOOL.BOX 1.0 }
```

Listing 2

```
Program PD;
{ Author:  Joe Wright
  Date:    22 Sep 89
  Version: 0.1

  Poorman's version of PWD.COM to demonstrate NZ-TOOL.BOX and
  access to the Z3NDIR structure.
}

{$I nz-tool.box}

Var NDIR : ndrptr;
    X, Y : Integer;

Begin
ndir := Ptr(getndr);
name := namstr(z3eadr^.extfcb^.name);
y := 7;
x := 25;

ClrScr;
WriteLn('Print Working Directories');
if fcb1.name[1] = '/' then
  Begin
    WriteLn('    Syntax:  ',name,' [P]');
    WriteLn('  The P option will show Passwords if Wheel is ON');
  End
else
  Begin
    Write(' Z3 Wheel is O');
    if getwhl then WriteLn('N') else WriteLn('FF');
    if ndir^.du <> 0 then
      Repeat
        if x > 20 then x := 1 else x := 35;
        y := y+1;
        GoToXY(x,y div 2);
        if getduok then pdu(Swap(ndir^.du));
        name := namstr(ndir^.name);
        pass := namstr(ndir^.pass);
        x := x+5;
        GoToXY(x,y div 2);
        Write(name);
        if (fcb1.name[1]='P') and getwhl then
          Begin
            x := x+9;
            GoToXY(x,y div 2);
            LowVideo;
            Write(' Pass: ',pass);
            NormVideo;
          End;
        WriteLn;
        ndir := Ptr(Ord(ndir)+SizeOf(ndrrec));
      Until Lo(ndir^.du) = 0
  End;
End.



{ Program:  CPY.PAS
  Author:   Joe Wright
  Date:     10 Sept 89
```

(Listing 2 continued on next page)

(Listing 2 continued from previous page)

```
  Version:  0.1

  This single-file copy program uses the NZ-TOOL.BOX to add
  many Z3LIB functions to Turbo Pascal programs.

  Version 0.2 is modified for the changes in NZ-TOOL.BOX

}

Program CPY;
{$I nz-tool.box}
{ Constants are used very much like EQUates. }
Const
  ver  = 0.2;     { Version Number }
  cr   = ^M;
  lf   = ^J;
  bel  = ^G;
  recs = 128;     { Sector size }
  bufs = 128;     { 128 of them for 16k (Max) }
Var
  inpbuf : Array[1..bufs,1..recs] of Byte;
  count  : Integer;
  i      : Integer;
  cks    : Integer;  { Checksum variable }
  ch     : Char;     { Keyboard response }
  o      : Boolean;  { Overwrite option  }
  n      : Boolean;  { No verify option  }

BEGIN {ning of CPY.PAS}
  name := namstr(z3eadr^.extfcb^.name);
  if fcb1.name[1] < '0' then {Help}
    Write(
      name,' Ver ',ver:1:1,' Single-File Copy Program',cr,lf,
      '  Syntax:  ',name,' [dir:]source [[dir:]destination] [/
oo]',cr,lf,
      '     where ''/oo'' is [O]verwrite and/or [N]o Verify options.'
      )
  else
    begin {source}
      o := false;
      n := false;
      curdu := retud;        { retud from nz-tool.box }
      dstdu := curdu;
      i := Pos('/',tbuff);  { check for '/' on command line }
      if i <> 0 then
        Repeat
          i := i+1;
          if tbuff[i] = 'O' then o := true;
          if tbuff[i] = 'N' then n := true;
        Until tbuff[i] = #0;
      spec1 := ParamStr(1);
      parse(spec1);          { parse dir: part to dirs }
      srcdu := dnscan;       { dnscan resolves d:, u:, du: and dir: }
      logud(srcdu);          { log it in }
      Assign(source,spec1);
      {$I-}
      Reset(source);         { open the input file }
      {$I+}
      if IOresult <> 0 then
        Write('Can''t find ',spec1)
      else
        begin {destin}
          spec2 := spec1;
          spec1 := ParamStr(2);
          parse(spec1);
          if (Length(spec1) = 0) or (spec1[1] = '/') then
            spec1 := spec2;
          dstdu := dnscan;
          if (dstdu = srcdu) and (spec1 = spec2) then
            Write(bel,'Don''t be silly..') { source=destination }
          else
            begin {copy}
              logud(dstdu);  { Log into destination area }
              Assign(destin,spec1);
              if not o then
                begin
                  {$I-}
                  Reset(destin);
                  {$I+}
                  If IOresult = 0 then
                    begin
                      pdu(dstdu);
                      if dutdir(dstdu) then Write(name,' ');
                      Write(spec1,' Exists.  Overwrite? (Y or N) ');
                      Read(Kbd,ch); WriteLn(ch);
                      if UpCase(ch) <> 'Y' then Halt;
                    end;
                end;
              ReWrite(destin);  { erases any existing file }
              { Tell the user we're up to something }
              Write(' Copying ');
              pdu(srcdu);
              if dutdir(srcdu) then Write(name,' ');
              Write(spec2,' to ');
              pdu(dstdu);
              if dutdir(dstdu) then Write(name,' ');
              WriteLn(spec1);
              cks := 0;  { clear the checksum word }

              Repeat
                Write('.');
                logud(srcdu);
                BlockRead(source,inpbuf,bufs,count);
                if not n then
                  for i := 0 to (count*recs)-1 do
                    cks := cks + Mem[Addr(inpbuf)+i];
                logud(dstdu);
                BlockWrite(destin,inpbuf,count);
              Until count = 0;

              Close(destin);
              if not n then
                begin {verify}
                  Reset(destin);   { Prepare to read it }
                  Write(cr,lf,' Verifying ');
                  pdu(dstdu);
                  if dutdir(dstdu) then Write(name,' ');
                  WriteLn(spec1);

                  Repeat
                    Write('.');
                    BlockRead(destin,inpbuf,bufs,count);
                    for i := 0 to (count*recs)-1 do
                      cks := cks - Mem[Addr(inpbuf)+i];
                  Until count = 0;

                  if cks <> 0 then Write(bel,cr,lf,'  Failed!')
                end; {verify}
            end; {copy}
        end; {destin}
    end; {source}
END. {of CPY.PAS}
```

■

If You Don't Contribute Anything

Then Don't Expect Anything

TCJ is User Supported

Embedded Applications

Z80 Communications Gateway, Part 1

by Art Carlson

When someone mentions their computer, we expect to see a keyboard, CRT display, and disk drives. But embedded controllers are also computers, even though they may not have any of the usually expected peripherals. They also use unfamiliar processors with strange architectures. Leaving the secure world of prepackaged desk top computers for the do-it-yourself world of embedded controllers can be rather frightening—it's something like taking the training wheels off your bike.

The design and application of embedded controllers is very interesting, and it is one of the few areas which provide significant opportunities for small entrepreneurial businesses. There are also many employment and consulting positions available for people with embedded controller application expertise.

I feel that working with embedded controllers can be fun, useful, and rewarding. The purpose of this series is to encourage you to explore the world of embedded controllers with the least trauma. Here, we will cover the low-level basics and use the more familiar tools in order to tempt you to get started on some beginning, but yet useful, projects. Other more advanced projects using different processors and tools will be covered in other articles.

Selecting the level at which to start is a very significant problem with any how-to-do-it series. Where ever we begin will be too advanced for some, and overly simple for others. Hopefully we will cover most needs through a combination of different articles. I don't intend to spend a lot of time on boring theory and facts to memorize, but rather will concentrate on learning by doing. An old Chinese proverb says, "I hear and I forget, I see and I remember, I do and I understand." My goal is to get you to assemble and program some simple controllers so that you will feel confident enough to tackle more advanced projects of your own choosing.

Controllers versus Processors

One of the most confusing aspects of embedded controllers is understanding the difference between the chips used in our desk top microcomputers and the chips used in embedded controllers. Our microcomputers generally use a microprocessor such as a Z80 (CP/M); 8088, 80286, 80386, etc. (IBM PC series); or 68000, 68020, 68030, etc. (Apple Macintosh series). Embedded controllers frequently use a microcontroller such as an 8031, 8096, 68HC11, 6803, or Z8. But, it gets very confusing because many embedded controllers use microprocessor chips such as the Z80, 80186, and 68000. Also, desktop microcomputers use microcontrollers, in addition to the main microprocessor, for disk drives keyboards, etc. Intel made a wise move when they changed the name of their 1989 edition *Embedded Control Applications Handbook* to *Embedded Applications* for the 1990 edition. I'll probably follow their lead and use the term *Embedded Application*, and not be concerned with whether the chip is a processor or a controller. We'll let others spend their time and effort on debating the semantics while we spend ours on building and programming devices.

"I hear and I forget,
I see and I remember,
I do and I understand."

An old Chinese proverb

One of the first steps in designing an embedded application is to select the chip (actually the first step is to define the requirements, but let's assume that has already been done). In order to minimize confusion, I'm going to refer to the chip as the *processor* instead of using different terms for a microprocessor or a microcontroller.

For some applications with the main emphasis on monitoring and setting I/O line status a processor such as the 8031 is a good choice. Other applications with the main emphasis on calculations and data manipulation may be better served with a processor such as the 68000. You will rarely find an application which can not be forced to run on most of the common processors. The choice is made based on cost, space, familiarity, available tools, precedent, stubbornness, and other factors which also may (or may not) include the features most suitable for the application.

In general terms, the processors considered for applications with a lot of computing requirements have a very powerful set of commands with calculating and data manipulation instructions. The more recent computing type processors can address a large memory area, and are intended to work with large program and data areas. The processors primarily considered as controllers generally contain on-chip I/O lines, counter/timers, A/D, serial communications, and/or other features which are usually provided by separate peripheral chips for computer type processors. The controller processors are usually intended to work with very small memory areas, with a fixed (in ROM) program. The program memory (8048) may be as small as 1K of ROM and the data memory as small as 64 bytes (that's BYTES not KBYTES!). The differences will become more understandable as we work with the different processors in this and other articles.

Even though we hear a lot about the

16/32 bit 80X86 and 680X0 families which are so popular in the desk top computers, 8 bit processors still rule the embedded applications field—in fact they still produce a large volume of 4 bit processors for embedded applications.

Enough discussion, let's get started.

Z80 Communications Gateway

Embedded applications frequently require the use of intra system communications both during the development and for the end use. The RS-232 serial interface is widely used for this purpose, and the ability to understand and implement this interface is essential.

The purpose of this project is to provide an RS-232 communications link which can be used with various computers so that future projects can be interfaced to any computer with an RS-232 port. It will also provide the background needed to design other serial port implementations.

I selected the Z80 processor for a number of reasons: 1) I had the chips, the tools (assembler, debugger, etc.), and the data books. 2) I am familiar with the instruction set. 3) I felt that the serial port could be implemented with the least amount of programming by using the Z80 CPU and the Z80 SIO. 4) I wanted to compare the amount of programming time and hardware cost/space with a future 8031 implementation. 5) Many readers have and are familiar with the Z80 and its tools, and I want to make it easy for them to get started.

The Z80 CPU

The Z80 is a very versatile processor. It has 40 pins (see Figure 1), some of which will not be used in the first stage of this project. The Z80 instruction set provides some very useful communications features, such as INIR (input to memory from I/O port and increment pointer until byte counter is zero), and OTIR (output from memory to I/O port and increment address until byte counter is zero). The Z80 interrupt protocol is implemented in the peripheral chips (Z80 SIO, CTC, PIO, DMA) which takes much of the pain out of implementing interrupt driven systems. Debates on polled versus interrupt designs generate lively discussions (arguments?) and we'll delve into this in later articles. In general, interrupts are faster and reduce the processor load but require more hardware, while polled response is slower and uses more processor time but requires less

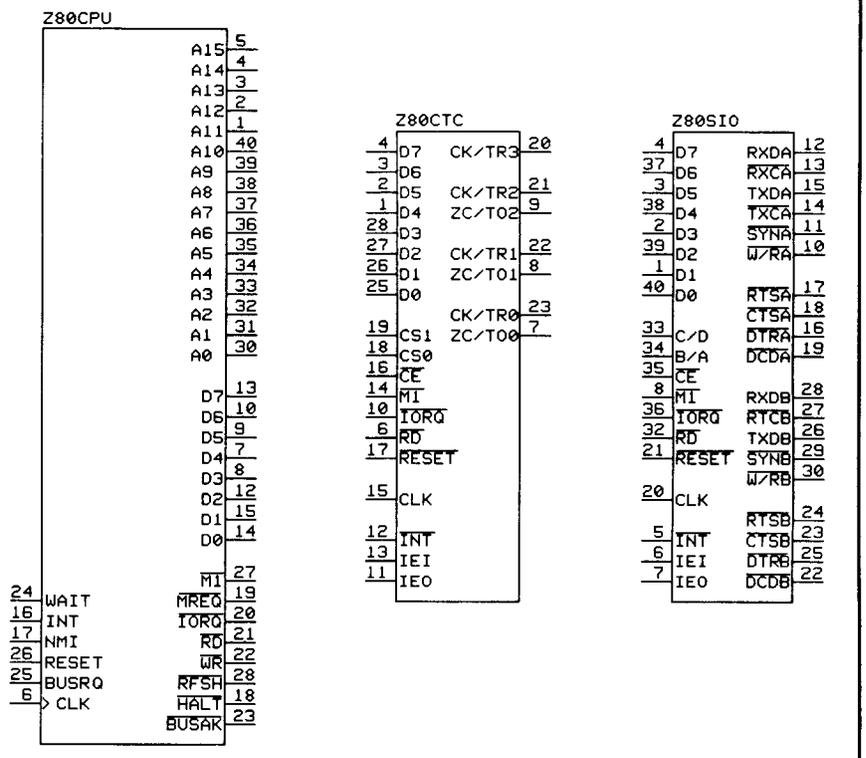


Figure 1: The Z80 Communications Chip set.

hardware. The Z80 is still currently in production, in fact a new 20 MHz version is now available. Technical literature is still available from Zilog, but many of the Z80 books from other publishers have gone out of print. We are preparing a Z80 resource directory which will list currently available reference material and tools.

The Z80 has 16 address lines which can address 65,536 memory locations. In computer terms this is referred to as 64K. The addressing capability is calculated by raising two to the power equal to the number of address lines (or address bits). A system with 8 address lines can address 2 raised to the 8th power, which equals 256 memory locations. Two to the 10th equals 1024, which we call 1K. Any system with 16 address lines has to use some sort of paging or memory management system (MMU) in order to address more than 64K (two to the 16th power).

The Z80 also has 8 bidirectional data lines which we will use to communicate with the peripheral chips and the serial port. Other pins will be discussed as they are used.

Buffers and Drivers

TTL and CMOC digital logic circuits employ high speed, low voltage, low current signals. These signals are too weak to operate relays, light LEDs, or to travel over wires off of the board. The Z80 can communicate with other TTL level chips through its address and data lines, but it needs help in order to communicate with off-board devices. The Z80 has address, data, and control lines, but it lacks the on-chip I/O lines which are on control oriented processors such as the 8031.

Peripheral chips are used to convert low-level logic signals to the higher levels needed for communications and control. There are many peripheral chips with different combinations of characteristics available. For this project we will use the MAX232 Dual RS-232 Receiver/Transmitter which is specifically designed for RS-232 serial communications. Buffers, drivers, latches, opto-islolators, and other peripheral chips will be covered in future projects.

Not all peripheral chips are used for signal level interfacing. There are also timers, counters, communication protocol, A/D and D/A, and many more devices. The

Z80 communications gateway uses the Z80 SIO, and the Z80 CTC. If we add a bidirectional parallel port we'll use the Z80 PIO.

RS-232 Serial Communications

The two methods of communicating between boards and/or systems are serial and parallel. With serial communications the information is sent through one pair of wires, one bit at a time, with each bit following the other. Parallel communications uses one wire for each bit, and sends a group of bits at the same time. Examples of serial communications are RS-232, RS422, and RS485. Parallel communications is used for the Centronics printer interface and for computer buses. Additional lines are often added for control signals.

Serial and parallel communications both have their uses. In general, serial is slower because the bits are sent one at a time, it takes more software and/or hardware to support the protocol, wiring costs are less because of the fewer wires required, and the standard serial implementations can handle long distances. Parallel is faster because more bits are sent at the same time (8 bits at once for Centronics, and up to 32 bits at once for some buses), it takes less software and/or hardware to support the protocol, wiring costs are higher because of the large number of wires, and the usual implementations are intended for shorter distances.

Communications involves much more than just dumping some bits into one end of a wire and picking them up a the other end. Communications protocol specifications can be broken down into two general areas. 1) Mechanical and electrical specifications, such as how many signal lines, voltage levels, connector design, etc. 2) Signal definitions, such as the number of bits, headers, control signals, timing, error detection and correction, etc. While the RS-232 interface is widely used, it is far from being standardized, and almost every manufacturer redefines the signal lines -- I have a rack full of adapter cables needed to make systems talk to each other. Many starting consultants paid the rent by trouble shooting RS-232 computer/printer interfaces with a breakout box. I'll design the gateway to interface with my equipment with a rather simple protocol. You may have to either modify it to work with your system, or else write compatible drivers for your system.

The following information is NOT a complete tutorial on RS-232--that would take a small book. Hopefully, it will be enough to get you started. I am considering a separate RS-232 tutorial booklet, and would like to hear how many people are interested.

RS-232 Electrical and Mechanical Specifications

RS-232 was written for two computers talking to each other over telephone lines with modems connecting the computers to the phone lines. It was intended for a DTE-DCE-telephone line-DCE-DTE link where each computer (the DTE, or Data Terminal Equipment) was connected to a modem (the DCE, or Data Communications Equipment), and the modems were connected to each other by the telephone line. The standard is too general, was not written with micros in mind, and is unnecessarily complex for most micro applications. When designers simplify the standard for their application, they don't choose the same path. This creates a confusing mess.

The RS-232 electrical specifications are quite complex. It includes things such as:

The driver (sending chip) must not be damaged by an open circuit or by a short to ground or to any other wire in the cable.

The terminator (receiving chip) must be able to tolerate 25 volts above ground and 25 volts below ground.

The driver's impedance should be selected such that the "one" and zero" levels at the output of the driver are between 5 and 15 volts in magnitude.

A logical "one," or MARK or OFF condition exists when the voltage at the interface point is between -3 and -15 volts.

A logical "zero," or SPACE or ON condition exists when the voltage at the interface point is between +3 and +15 volts.

The RS-232 driver and electrical include bipolar (both positive and negative) voltages and voltage levels above those that TTL logic chips can supply. There are standard driver and receiver chips for RS-232, and everyone I know uses these chips instead of trying to clobber up something. The industry standard chips are the 1488 Quad line driver and the 1489 Quad line receiver (49 cents each at Jameco). The

Pin	Mnemonic	RS-232	Description	Direction
			Ground	
1	GND	AA	Protective Ground	X
7	GND	AB	Signal Ground	X
			Data	
2	TXD	BA	Transmitted Data	From DTE
3	RXD	BB	Received Data	To DTE
14	(S)TD	SBA	Secondary Transmitted Data	From DTE
16	(S)RD	SBB	Secondary Received Data	To DTE
			Control	
4	RTS	CA	Request to Send	From DTE
5	CTS	CB	Clear to Send	To DTE
6	DSR	CC	Data Set Ready	To DTE
8	DCD	CF	Data Carrier Detected	To DTE
20	DTR	CD	Date Terminal Ready	From DTE
21	SQ	CG	Signal Quality Detector	To DTE
22	RI	CE	Ring Indicator	To DTE
23		CH	Data Signal Rate Selector (DTE)	From DTE
23		CI	Data Signal Rate Selector (DCE)	To DTE
12	(S)DCD	SCF	Secondary Data Carrier Det.	To DTE
13	(S)CTS	SCB	Secondary Clear to Send	To DTE
19	(S)RTS	SCA	Secondary Request to Send	From DTE
			Timing	
15	(TC)	DB	Trans. Signal Element Timing (DCE)	To DTE
17	RC	DD	Rec. Signal Element Timing (DCE)	To DTE
24	(TC)	DA	Trans. Signal Element Timing (DTE)	From DTE
			Miscellaneous	
9			Reserved for Data Set Testing	
10			Reserved for Data Set Testing	
11			Unassigned	
18			Unassigned	
25			Unassigned	

Figure 2: RS-232 signal descriptions

driver requires both positive and negative voltages, and is usually supplied with +12 and -12 volts. Most small computer boards will run on just +5 volts, except for the RS-232 port, and it was aggravating to have to provide +12 and -12 volts for the serial port. Maxim developed the MAX232 chip with two receiver and two driver channels plus charge pumps which runs on just the usual +5 volts and generates +10 and -10 volts on the chip— you'see this chip used frequently here and in Circuit Cellar Ink. Motorola has recently announced the MC145407 5 volt only Driver/Receiver with three drivers and three receivers. I've been told that on occasion the Maxim chip lacks the power to drive long lines, and in these cases the Motorola chip has provided enough power to solve the problem.

The standard defines 25 pins, three of which are unassigned and two of which are reserved for testing (see Figure 2), but does not specify the size or shape of the connector. For a long time the de facto standards were the DB-25S (female) and DB-25P (male connectors. IBM uses a non-standard nine pin DE9 connector for the serial port and an DB25 for the parallel port. Some manufacturers have put both a serial and a parallel port on a single DB25 connector, while others have added a current loop interface to an RS-232 DB25 connector—you'll have to watch out for non-standard bastard implementations.

You really need the RS-232 specifications for any computer, terminal, printer, or other device with which you want to communicate. My Morrow Decision I S-100 systems are non-standard, and I had to learn a lot about RS-232 before I could get them to communicate. The Ampro Z80 Little Board (which is now again available, see editorial), is more standard, using six wires (Protective Ground, RXD, TXD, RTS, CTS, and Signal Ground).

The minimum for two way RS-232 communications is three wires with Signal Ground, RXD, and TXD, which is what is provided on most controller processors.

The formal name of the RS-323 is *Interface between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange*. It describes, 1) The mechanical description of interface circuits, 2) The functional description of interchange circuits, 3) The electrical signal characteristics. It does not specify the ASCII character set or how to control message transfer using ASCII. We need a new up-to-date standard, one author has referred to RS-232 as *death throes of a dinosaur*. But RS-232 is what we have, and we'll have to make it work.

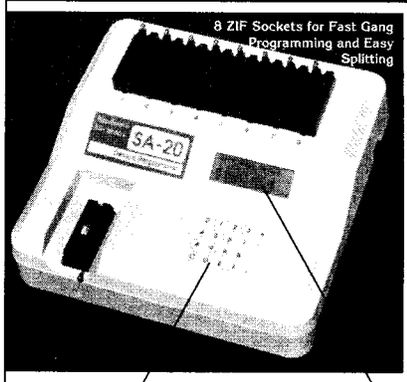
Serial Binary Data Transfer using a UART

The subject of data transfer is entirely separate from the subject of RS-232, although we usually consider them as one and the same. But RS-232 only describes the lines and voltages, it doesn't specify what is being sent. A full discussion of start bits, data bits, stop bits, and parity would take most of TCJ. We'll leave that to the bit-bangers for now, and use the Z80 SIO UART (Universal Asynchronous Receiver Transmitter) which does most of the hard work for us. The SIO is configured by writing to certain registers during system initialization, and then the parallel data is sent to and received from the SIO.

The SIO BAUD rate (used in honor of a Frenchman named Baudot) is determined by the combination of the receiver and transmitter clock signals and the clock mode. The Z80 CTC Counter/Timer is used to provide programmable clock rates for the SIO.

EPROM PROGRAMMERS

Stand-Alone Gang Programmer **$750.00**



- Completely stand-alone or PC driven
- Programs E(E)PROMs
- ***1 Megabit of DRAM***
- ***User upgradable to 32 Megabit***
- ***.3/.6" ZIF socket, RS-232, Parallel In and Out***
- 32K internal Flash EEPROM for easy firmware upgrades
- ***Quick Pulse Algorithm (27256 in 5 sec, 1 Megabit in 17 sec.)***
- 2 year warranty
- Made in U.S.A.
- Technical support by phone
- Complete manual and schematic
- ***Single Socket Programmer also available. $550.00***
- Split and Shuffle 16 & 32 bit
- 100 User Definable Macros, 10 User Definable Configurations
- Intelligent Identifier
- Binary, Intel Hex, and Motorola S

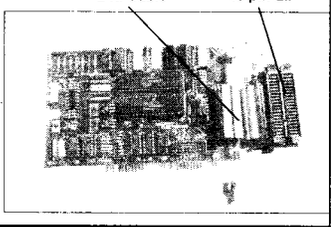
20 Key Tactile Keypad (not membrane) 20 x 4 Line LCD Display

Internal Programmer for PC **$139.95**

New Intelligent Averaging Algorithm. Programs 64A in 10 sec., 256 in 1 min., 1 Meg (27010, 011) in 2 min. 45 sec., 2 Meg (27C2001) in 5 min. Internal card with external 40 pin ZIF.

- Reads, verifies, and programs 2716, 32, 32A, 64, 64A, 128, 128A, 256, 512, 513, 010, 011, 301, 27C2001, MCM 68764, 2532
- ***Automatically sets programming voltage***
- Load and save buffer to disk
- Binary, Intel Hex, and Motorola S formats
- ***Upgradable to 32 Meg EPROMs***
- ***No personality modules required***
- 1 year warranty • 10 day money back guarantee
- Adapters available for 8748, 49, 51, 751, 52, 55, TMS 7742, 27210, 57C1024, and memory cards
- Made in U.S.A.

2 ft. Cable 40 pin ZIF



NEEDHAM'S ELECTRONICS
4539 Orange Grove Ave. • Sacramento, CA 95841
Mon. - Fri. 8am - 5pm PST
C.O.D.

Call for more information
(916) 924-8037
FAX (916) 972-9960

Next Time

So far we have concentrated on the fundamentals. Next time we'll cover the details of the Z80 SIO and CTC, and how they are used for RS-232 serial communications. If you have any questions or comments on RS-232 or serial communications, send them for use in this column. •

References

The RS-232-C Serial Interface, by Phil Wells, Parts One and Two, TCJ issues Number 1 and Number 2.

Advanced CP/M

Tuning JetFind

by Bridger Mitchell

Trenton Computer Festival Notes

This year, the 15th annual Trenton Computer Festival featured an all-day Saturday section on current and future Z-System developments. Coordinated by Jay Sage, the program included:

10:30 Bridger Mitchell: Tools for CP/M—DosDisk, JetFind, PluPerfect Writer, and the new ZMATE editor.
11:45 Bob Todd: SIG/M: Future of Public Domain Software.
1:00 Jay Sage: Introduction to Z-System.
2:15 Hal Bower, Cam Cotrill: The Future of 8-Bit Operating Systems.
3:30 Jay Sage: Computing Automation Using ARUNZ Scripts.
4:20 Al Hawley: Assembly Language Programming and the New ZMAC Assembler.
4:45 Bridger Mitchell: Multitasking with BackGrounder ii.

Despite the heavy rains on Saturday, it was standing room only for several talks, and conversations continued well into the morning hours of Sunday. Then it was catch a few winks and off to the flea market.

As a first-time attendee I particularly enjoyed meeting so many talented Z-System colleagues face to face! Cam Cotrill, Al Hawley and Rob Friefeld also came from Los Angeles, and Carson Wilson from Chicago. Others, from up and down the East Coast, included Lee Bradley, Howard Goldstein, Liv Hinckley, Chris McEwen, Bruce Morgen, Dick Roberts and Bob Schultz. If you missed Trenton, look out for Tony Parker's "z-system, lies, and videotape"!

Tuning JetFind

Computers are good for organizing information—or at least they should be! It's easy enough to save correspondence, notes, old programs, addresses and most everything else that clutters up a real desk in files on disks. The files can have somewhat meaningful names, and related files can be stored together in library files. And disk space can be reduced by crunching or squeezing the files first.

The rub comes when you need to retrieve some of that information. Much of the time I can't remember exactly which file I saved it in. And in many cases I need only a few lines of material buried inside the file. Or, I may be looking for a particular subroutine or data structure.

Some three years ago I looked into the programs available to handle these retrieval tasks. Dissatisfied with them all, I wrote JetFind, a tool that can find one or several *strings* in files.

Three things set JetFind apart. First, it works with most types of CP/M files—text, WordStar, crunched, squeezed, and also any of those types when stored a library. Second, it has very powerful "wild-card" capability and can match *regular expressions.* And third, it is *fast,* so fast that many users find it more convenient to search for information by running JetFind than to reach up to their bookshelf for documentation that is near at hand. Comparative testing by others has established that JetFind is some two to six times faster than other Z80 search tools.

String Searches

Searching for strings is one of those programming tasks that presents itself as a really standard, top-to-bottom problem, one that almost automatically organizes itself into logical blocks. Once you have obtained the search parameters from the user, your program's tasks are:

```
open a file
  read a block
    move a line into linebuffer
      scan line for pattern
        if found, report success
```

In each task block, the task is repeated until a termination condition is encountered ("while more files", ..., "while another pattern"). Like the poet's fleas, a smaller task is perched on the shoulders of its surrounding task block. Indeed, if we look inside the scanning block, it too has still smaller "while" loops.

With this strong hierarchy governing the search, it's pretty clear that there are big payoffs to tuning the innermost task—the line scanner that matches a pattern. And, indeed, a better pattern-matching algorithm proves to be a better mousetrap.

What is not so obvious, perhaps, is that simply getting a byte of input from a file is also a place where major speed can be gained. In developing JetFind I had paid a good deal of attention to both areas, and achieved some significant speedups over the "obvious" hierarchy.

---

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

A New Burst of Speed

Despite JetFind's good performance, since completing the distribution version three years ago I've occasionally wondered if I had overlooked some opportunity for still-faster searching. And I've kept watch on some of the computer science literature that covers efficient searching algorithms.

Recently, two papers caught my attention. Andrew Hume, (in the November 1988 *Software—Practice and Experience*) describes his success in speeding up two UNIX versions of 'grep'. Using a profiler, he isolated time-intensive sections of the code. Replacing the standard C input/output library functions more than tripled the speed of grep. That result didn't surprise me; in JetFind I had worked hard to optimize the file input routine, and had already won substantial gains. But his second optimization held still unrealized potential for JetFind -- using a variant of the "Boyer-Moore" algorithm.

Smart Searching

Everyone knows how to search for a pattern by "brute-force." You start at the beginning of the line and compare the first byte with the first byte of the pattern. If they match, compare the second bytes, etc. If all pattern bytes match, the search succeeds. If not, point at the second byte of the line and restart the pattern comparison.

In 1977 (*Communications of the ACM*, October) Boyer and Moore published their key idea—start comparing at the *end* of the pattern, and work backward toward the first byte. Then, when a mismatch occurs, move forward as far as possible and then again compare from the end of the pattern.

The Boyer-Moore algorithm works like this. Suppose we are searching for the word "where". It has 5 characters, one of which is repeated. We make the first check at position 5 of the line, comparing the final pattern letter, an 'e', with the letter at that position in the text being searched.

```
          1 2 3 4 5 6 7 8 9 0
text:     w h e r e   i s
pattern:  w h e r e
```

If it is 'e', we check the previous character for a match with 'r'. And we continue checking from right to left until either we have matched all 5 characters and found an instance of the pattern (shown above) or we have a mismatch.

Most of the time a mismatch occurs, and then we jump ahead, by sliding the pattern to the right as far as possible.

```
          1 2 3 4 5 6 7 8 9 0
text:     e v e r y w h e r e
pattern:  w h e r e
```

How far can we slide? Suppose, first, that the first text character that we check (a 'y') doesn't occur anywhere in the pattern. In this case we can slide the pattern 5 characters to the right, the length of the pattern.

```
          1 2 3 4 5 6 7 8 9 0
text:     e v e r y w h e r e
pattern:  . . . . . w h e r e
```

Figure 1. Sketch of Boyer-Moore Algorithm in Z80 Code.

```
more:   ld      de,(text.)      ; point at right end of candidate text
        ld      hl,(pat.)       ; point at right end of pattern
        ld      a,(line_length)
        cp      e
        jp      c,not_found
;
loop:   ld      a,(de)          ; compare text with pattern
        cp      (hl)
        jp      nz,next
        dec     l               ; back up text ptr
        dec     e               ; back up pattern ptr
        jp      p,loop          ; while more pattern bytes, loop
        inc     e               ; point at beginning of match
        jp      found
;
next:   ld      d, HIGH Dtable  ; prepare pointer to displacement table
        ld      hl,(text.)      ; point at right end of candidate text again
        ld      e,(hl)          ; get right text character, use as index to table
        ld      a,(de)          ; get displacement value from table
        add     l               ; add it to text pointer
        ld      (text.),a       ; update low byte of pointer
        jp      more            ; and do next comparison

;
text.:  dw      linebuf + patlen-1
pat:    dw      patbuf  + patlen-1

        org     200h
linebuf:
        org     300h
patbuf:
        org     400h
Dtable:
```

Figure 2. Initializing the Displacement Table

```
MAXSYM  equ     128             ; input chars have been masked to 7-bits
                .new
        ld      b,MAXSYM        ; set all displacements
        ld      hl,Dtable       ; to pattern length
        push    hl
        ld      a,patlen
1$:     ld      (hl),a
        inc     hl
        djnz    1$
;
        pop     hl
        dec     a               ; patlen-1
        ld      b,a
        ld      de,patbuf       ; -> 1st byte of pat
        ret     z               ; if 1-byte pattern, skip loop
2$:     ld      a,(de)          ; get pattern byte
        ld      l,a             ; point at that character in displacement table
        ld      (hl),b          ; install distance to jump
        inc     de
        inc     hl
3$:     djnz    2$
        ret
```

Why? Because now that we know that the pattern cannot be found in the first 5 characters of the text because they include something not in the pattern.

Suppose, instead, that the first character we check (a 'w') does occur in the pattern, but not as the last character.

```
          1 2 3 4 5 6 7 8 9 0
text:     n o w   w h e r e
pattern:  w h e r e
```

Then we can slide the pattern to the right far enough to align that position in the text with the pattern, and then again start checking from the right end of the pattern. (In this case we slide right by 4 characters).

```
          1 2 3 4 5 6 7 8 9 0
text:     n o w   w h e r e
pattern:  . . . . w h e r e
```

Boyer-Moore is a kind of leap-frog algorithm. It has the directness and impact of a light-bulb-just-turned-on, why-didn't-I-think-of-that! An idea that is "obvious" in retrospect, and a contribution you will

always admire.

Unfortunately, the details aren't equally direct. The algorithm and several variations have been studied in considerable depth in a succession of computer science papers. A number of actual implementations have also been described.

The second article I had spotted, in the July 1989 issue of *Dr Dobbs' Journal*, claimed a major improvement in search speed implementing the Boyer-Moore method under MS-DOS TurboPascal. But on closer investigation a reader had found (*DDJ*, October 1989) that particular code proved to be slower than brute force coded in the same language!

Nevertheless, I kept returning to the Boyer-Moore algorithm in idle moments, feeling there must be some way to exploit its inherent efficiency. And ultimately I found it.

The Z80 Algorithm

For this implementation I could assume that the text to be searched was a line already in a buffer and not more than 254 bytes long. I put the pattern in a second buffer and arranged the buffers so that both begin on "page" (nn00 hex) boundaries.

Look at Figure 1. The heart of the search is quite elegant. Short, quick instructions manipulate the low byte of the pattern and text pointers.

The major piece missing from Figure 1 is the displacement table that tells the algorithm how far to shift the pattern, or equivalently, how many characters may be jumped over. Figure 2 gives a simple routine for initializing it. First, all displacements are set to the length of the pattern, pretending that each of those characters does not appear in the pattern anywhere. Then, for each character that does appear in the pattern, the displacement is "corrected" to be the distance from that character to the end of the pattern.

Because this "correction" value is computed from left to right, it will never be too large and cause the jump to skip past a true match. In some circumstances, however, it results in a smaller than optimal jump. The optimal algorithm been worked out, but it is more complicated and also increases the number of instructions in the innermost comparison loop. The simple correction algorithm I used is, I believe, due to Horspool (1980 *Software—Practice and Experience*).

Actual Implementation

Getting a search algorithm working in a real program requires paying attention to a host of other details. You have to initialize the pointers and line length before you can use the routine. And of course, you have to have all of the surrounding code from the higher-level blocks, as well as the user interface, to specify the patterns, files, and options. In all, JetFind is some 16K.

Moreover, it sometimes pays to look before you leap-frog with Boyer-Moore. You can scan the line for the first occurrence of the final character of the pattern, using the zippy Z80 CPIR instruction. This quickly rules out many lines that don't contain the pattern, and gets the search started well into the line for others. However, because this involves inspecting each character, it can be less efficient than using just the Boyer-Moore search when the pattern is more than a few bytes long.

The new version of JetFind checks to see if the pattern is a "plain" pattern, or includes one or more wild-card characters that make up a general regular expression. If it is plain, it uses the Boyer-Moore algorithm. If not, it uses the already developed "grep" algorithm.

There's one other point of some interest. When a crunched file is processed, some of the logic is inverted. You might think that uncrunching would simply involve adding an "uncrunch-block" task between steps 2 and 3. And it could be done that way. But decompression is inherently a variable-flow process: you read some bytes and bits and produce some, or perhaps quite a few more, as output. So, instead of managing a variable-sized buffer for the uncrunched output, it is more convenient to invert the process. The uncruncher takes control, reading blocks and calling the pattern-scanner for each line, until the file is fully processed. By maintaining well-modularized routines, it was possible to introduce this change in flow with very little recoding.

```
open a file
 call uncruncher
  read a block
   uncrunch a line
    move line into linebuffer
     scan line for pattern
      if found, report success
```

Performance

The theoretical performance of the Boyer-Moore algorithm is impressive. It's worst-case running time is proportional to the number of bytes in the input text ("linear"), and its average time is less than proportional ("sub-linear").

To get an approximate measure of the improvement obtained, I ran the old and new JetFind versions on the source files (147K in all), using timing to the nearest second that is automatically reported by JetFind when running on a DateStamper system.

For this test the basic overhead to read the files was 5 seconds on an SB180FX's ram disk (at 9 MHz, 1 wait state). Searching for four identical patterns ("expression") , JetFind version 1.22 required an additional 7 or 10 seconds, depending whether case folding was requested when it used the "grep" algorithm. Using the Boyer-Moore algorithm, the extra time was just 2 or 3 seconds. A single pattern can be searched in essentially the time required to read the files!

	JetFind 1.22	JetFind 1.28 with Boyer Moore
read files	5	5
1 pattern	6	5
ignoring case	7	5
2 patterns	9	6
ignoring case	10	6
4 patterns	12	7
ignoring case	15	8

Other Search Algorithms

The search for a still more efficient search (!) has occupied more than a few good minds, and there is a growing literature on variants of Boyer-Moore and other algorithms. Significant gains can be achieved for special cases such as small alphabets and large memories. The vanilla Boyer-Moore, however, is likely to remain the speed leader for general-purpose string searching. •

Animation With Turbo C Ver. 2.0

Part 2: Screen Interactions

by Clem Pepper

Before beginning new topic I must back up a couple issues to make a correction. I hereby apologize for any frustration resulting from my omitting the variable declarations from some programs in the March/April issue. Also, after emphasizing the use of HEADING.C I am replacing it with a superior approach. The programs I am referring to are RECT1.C, PLANE.C, TANK.C and VIEWPORT.C. Listing 1 contains the revised declarations.

Listing 2 is an include file, GRAP_ENB.H. Its use is illustrated in the revisions of Listing 1. All the required function calls for enabling the graphics mode are included in this file. The palette number is declared in the program file and passed to GRAP_ENB.H in the function call **enable_graph(palette);**. Wish I had thought of this at the beginning, it is so much cleaner.

Taking Advantage Of Turbo C's MAKE Utility

In this segment of the series on animation we are taking our initial step toward the production of a complete screen action game. This portion of the game consists of five modules. The five require compiling and linking to acquire the .EXE run file. Normally when in the process of developing a project of this scope I give each of the modules its own **main()** so it can be compiled and run independently. Using that approach with this number of modules in an article series is impractical, so we sort of have to dive right in.

It's not all that bad. Learning to use MAKE straight out of the Borland manual can be frustrating; it took me the better part of three evenings to make sense of it. So you're getting the benefit of my frustrations—be happy.

MAKE really makes compiling and linking modules easy and fast. The MAKE file for the game we are going to learn about here is shown in Listing 3. The top line of the listing is where the action all comes together at the finale. It defines the .EXE file as the end product of the module .OBJ files. The next line calls TLINK for the module linking at the .OBJ level. This instruction is spread over two lines as a necessity for printing. It MUST ALL BE ON ONE LINE in your .MAK file. The lines below perform the compiling of the individual C modules. The sequence doesn't matter, though I normally place the module containing **main()** at the bottom.

My name for this file is TNK_WAR.MAK. You can call it by any name you choose so long as the extension is .MAK. The command line must read MAKE -FTNK_WAR (in my case). Without the leading -F the program will abort.

You may very well need to make changes from the listing. I do all my work on a floppy in drive A:. My Turbo C files are on a hard drive, directory E:\btc20, E:\btc20\include, and E:\btc20\lib. My AUTOEXEC includes a path statement of:

```
PATH=C:\PCWRT;E:\BTC20
```

Listing 1. Corrections to the previous graphics program listings.

```
            ************
RECT1.C
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{ int palette = 2;  /* specify 0,1,2, or 3 */
   enable_graph(palette);
    |
     pick up from article listing.

            ************
PLANE.C
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{ int palette = 2;  /* specify 0,1,2, or 3 */
   enable_graph(palette);
    |
     pick up from article listing.

            ************
VIEWPORT.C
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{ int palette = 2;  /* specify 0,1,2, or 3 */
  int i = 9, left_col = 0; rite_col = 9, top = 0, bottom = 9;
   enable_graph(palette);
    |
     pick up from article listing.

            ************
TANK.C
#include <stdio.h>
#include <graphics.h>
#include "grap_enb.h"

/* == Begin program == */
main()
{ int palette = 2;  /* specify 0,1,2, or 3 */
   enable_graph(palette);
    |
     pick up from article listing.

            ************
```

Because I am still using MS-DOS 2.11 only .EXE files are identified. So I copy C0S.OBJ, TURBOC.CFG and GRAPHICS.LIB onto the floppy. So you must modify your MAKE file to conform to your system.

To use the MAKE utility with the five modules all that is necessary is to type up the .C source files. With this done simply enter **make -ftnk_war** on the command line, press the Enter key, sit back and enjoy the clicking as your hard disk and floppy go through the process of compiling and linking the modules. The real beauty of it is when you make a change in one of the modules. MAKE identifies the changed module by comparing the time and date on the source and object files. Really neat!

Listing 2. The header file for enabling the graphics mode.

```
/* GRAP_ENB.H
** Call this function with the main() module of a graphics program.
*/

/* == enable the graphics mode == */
enable_graph(int graphmode)
{
int graphdriver = CGA;       /* graphics driver       */
int errorcode;               /* graphics error code   */

initgraph(&graphdriver, &graphmode, "e:\\btc20");
  /* ** replace "e:\\btc20" with your directory location ** */
errorcode = (graphresult());      /* get result code */

  /* ** graphics error function routine call ** */
   if (errorcode != grOk)   /* always check for error */
  {
    printf("Graphics error: %s\n",grapherrormsg(errorcode));
    exit(1);
   }

  /* ** call to set background color ** */
  setbkcolor(BLUE);
}
```

Listing 3. The Turbo C MAKE file for TANK_WAR.C.

```
tank_war.exe: tank_war.obj bomr.obj shell.obj gme_figs.obj bombs.obj
  tlink c0s bombs gme_figs shell bomr tank_war,tank_war,/x,
    e:\btc20\lib\cs graphics.lib  Note: This MUST all be on one
line.

bombs.obj: bombs.c
  tcc -a -c -ms bombs.c

gme_figs.obj: gme_figs.c
  tcc -a -c -ms gme_figs.c

shell.obj: shell.c
  tcc -a -c -ms shell.c

bomr.obj: bomr.c
  tcc -a -c -ms bomr.c

tank_war.obj: tank_war.c
  tcc -a -c -ms tank_war.c
```

The Basics of Module Construction

The five modules of Listings 4 - 8 are printed with line numbers. This to make it easier for me when referring to a specific function or whatever in any given module.

Starting at the top we observe that each module contains only those #include files required for itself. The primary module, TANK_WAR.C is loaded with the most, seven. The majority of the remainder have only one or two. A mandatory #include <graphics.h> is found in each. The other is <stdio.h>, required when we use some of the keyboard and console functions.

There are a lot of global declarations. The preferred technique with C is to pass values in function calls, but this gets out of hand quickly in this kind of a program. So we take the easy way out.

When a module requires a unique variable that will be used by one or more of the other modules it is declared in the originating module in the usual fashion as a global int or char or whatever. In those modules making reference to that variable the declaration is preceded by **extern**. This informs the compiler, TCC, that the variable is declared somewhere else. We don't have to be specific on where somewhere else is, the compiler will track that down on its own very nicely.

To be fair I should mention I do all my work with my own editor, PC-Write and the command line. I almost never use the environment file, TC. That should not in any way effect the results obtained with TANK_WAR as shown in the listings.

In one respect the modules are rather sloppy in that I have not been too concerned with function prototypes. If you want to work them in, feel free.

Listing 4. The main module for the TANK_WAR action game.

```
 1: /* TANK_WAR.C
 2: ** Main module for TANK_WAR screen action game.
 3: ** Compiled with TURBO C Ver. 2.0 graphics library routines.
 4: **
 5: */
 6: #include <stdio.h>
 7: #include <conio.h>
 8: #include <alloc.h>
 9: #include <ctype.h>
10: #include <dos.h>
11: #include <graphics.h>
12: #include "grap_enb.h"
13: #define SFKEY 0x16
14: extern void clr_bomrs();
15:   /* ** global declarations ** */
16: int n_asc, scn;
17: int leftt_col = 200, ritet_col = 219;
18: int tdir_flg = 0; /* tank moves to left */
19: extern char far *tank;
20: extern int bh1_hit,bh2_hit,bh3_hit;
21:   /* == read non-ascii key == */
22: int rd_nonasky()
23: {
24: union REGS regs;
25:   regs.h.ah = 0;
26:   regs.h.al = 0;
27:    int86(SFKEY, &regs, &regs);
28:     n_asc = regs.h.ah; /* non-ASCII code */
29:     scn = regs.h.al; /* SCAN/ASCII code  */
30: }
31:   /* == Begin program == */
32: main()
33: {
34: int palette = 2, i = 1;
35: char key, run = 0;
36:  enable_graph(palette);
37:   /* ** create game figures ** */
38:    drw_figs();
39:   /* ** set viewport for scoring ** */
40:    /* This will be added later */
41:   /* ** begin animation ** */
42:    while(i) {  scn = 1;
43:   /* ** tank direction on screen control ** */
44:          if(tdir_flg == 1)          {
45:          leftt_col += 3; ritet_col += 3;
46:                                      }
47:          else if(tdir_flg == 0)
48:          { leftt_col -= 3; ritet_col -= 3; }
49:             draw_tank();
50:     /* ** increment bomber group ** */
51:      bomber();
52:      if(bh1_hit == 0 || bh2_hit == 0 || bh3_hit == 0)
53:       drop_bombs();
54:       shel(run); run = 0;
55:       drop_bombs();
56:         clr_bomrs();
57:          putimage(leftt_col,189,tank,XOR_PUT);
58:    if(leftt_col <= 4) { tdir_flg = 1; continue; }
59:    if(ritet_col >= 297) { tdir_flg = 0; continue; }
60:  /* ** obtain input from the keyboard ** */
61:    if(kbhit() == 0) continue;
62:     rd_nonasky();
63:      if(n_asc == 72 || n_asc == 75 || n_asc == 77) goto L;
64:        else { run = toupper(toascii(scn));
65:           if(run == 'Q') { i = 0; break; }
66:      else if(run == 'A' || run == 'S' || run == ' '
67:       || run == 'D' || run == 'F') continue;
68:                          }
69:  /* ** move tank to the right ** */
70:     L:  if(n_asc == 77)
71:         { tdir_flg = 1; continue; }
72:  /* ** move tank to the leftt ** */
73:          else if(n_asc == 75) { tdir_flg = 0;
74:             continue;        }
75:  /* ** halt tank in current position ** */
76:          else if(n_asc == 72) { tdir_flg = 2;
77:             continue;        }
78:                }
79:       closegraph();  /* return to text mode */
80:        exit(0);
81: }
82:   /* == draw tank as sequence of horiz lines == */
83: draw_tank()
84: {
85:   putimage(leftt_col,189,tank,COPY_PUT);
86: }
```

As you scan through the listings take note of the large number of comments. Getting lost in the maze of program interactions between modules is no problem at all. In your modules comment

Listing 5. The TANK_WAR game module for play object construction.

```
/* GME_FIGS.C
** Game figure creation module for TANK_WAR game.
** Compiled with TURBO C Ver. 2.0 graphic library routines.
*/
#include <alloc.h>
#include <graphics.h>
  /* ** global declarations ** */
char far *tank;
char far *bomr;
char far *shell;
char far *burst;
char far *bomb;
  /* == begin program == */
drw_figs()
{
char buffer[80];
unsigned numbytes;
  /* ** draw initial tank figure ** */
    setlinestyle(0,0,1);  /* solid line, one pixel wide */
    setcolor(2);          /* tank top is red */
     moveto(9,0);  lineto(15,0); /* segment a */
    setlinestyle(0,0,3);  /* solid line, 3 pixels wide */
     moveto(7,2);  lineto(17,2); /* segment c */
    setlinestyle(0,0,1);  /* solid line, one pixel wide */
     moveto(9,4);  lineto(15,4); /* segment d */
    setcolor(1);          /* lower tank is green */
     moveto(5,5);  lineto(19,5); /* segment e */
     moveto(4,6);  lineto(19,6); /* segment f */
     moveto(5,7);  lineto(19,7); /* segment g */
     moveto(6,8);  lineto(18,8); /* segment h */
     moveto(7,9);  lineto(17,9); /* segment i */
  /* ** determine storage needed ** */
     numbytes = (unsigned int)imagesize(0,0,20,20);
  /* ** allocate buffer ** */
     tank = (char far *)malloc(numbytes);
     getimage(0,0,20,20,tank);  /* save the image */
      cleardevice();
  /* ** draw initial bomber figure ** */
    setcolor(2);          /* fuselage is red */
  /* ** draw wings ** */
    setcolor(1);          /* wings, tail are green */
     moveto(9,1);   lineto(9,8);  /* segment w1 */
     moveto(10,2);  lineto(10,8); /* segment w2 */
     moveto(11,3);  lineto(11,8); /* segment w3 */
     moveto(12,4);  lineto(12,8); /* segment w4 */
     moveto(13,5);  lineto(13,8); /* segment w5 */
     moveto(14,6);  lineto(14,8); /* segment w6 */
     moveto(14,12);  lineto(14,14); /* segment w7 */
     moveto(13,12);  lineto(13,15); /* segment w8 */
     moveto(12,12);  lineto(12,16); /* segment w9 */
     moveto(11,12);  lineto(11,17); /* segment w10 */
     moveto(10,12);  lineto(10,18); /* segment w11 */
     moveto(9,12);   lineto(9,19);  /* segment w12 */
  /* ** draw tail ** */
     moveto(1,5);   lineto(1,15);  /* segment s1 */
     moveto(2,5);   lineto(2,15);  /* segment s2 */
     moveto(3,6);   lineto(3,14);  /* segment s3 */
     moveto(4,7);   lineto(4,13);  /* segment s4 */
  /* ** draw fuselage ** */
    setcolor(2);          /* fuselage is red */
     moveto(8,8);  lineto(15,8); /* segment f1 */
     moveto(4,9);  lineto(18,9); /* segment f2 */
     moveto(0,10);  lineto(20,10); /* segment f3 */
     moveto(4,11);  lineto(18,11); /* segment f4 */
     moveto(8,12);  lineto(15,12); /* segment f5 */
  /* ** determine storage needed ** */
    numbytes = (unsigned int)imagesize(0,0,20,20);
  /* ** allocate buffer ** */
     bomr = (char *)malloc(numbytes);
     getimage(0,0,20,20,bomr);  /* save the image */
      cleardevice();
  /* ** draw shell figure ** */
  /* ** as a sequence of vert lines ** */
   setlinestyle(0,0,1);
   setcolor(3);   /* brown */
    moveto(0,2);  lineto(0,5);
    moveto(1,0);  lineto(1,6);
    moveto(2,2);  lineto(2,5);
  /* ** determine storage needed ** */
    numbytes = (unsigned int)imagesize(0,0,6,6);
  /* ** allocate buffer ** */
    shell = (char *)malloc(numbytes);
    getimage(0,0,6,6,shell); /* save the image */
     cleardevice();
  /* ** draw burst figure ** */
  /* ** as a sequence of lines ** */
   setcolor(2);  /* red */
   setlinestyle(0,0,1);
     moveto(4,4); lineto(4,7);
     moveto(5,4); lineto(5,6);
     moveto(6,3); lineto(6,7);
     moveto(7,4); lineto(7,6);
   setcolor(3);  /* brown */
     moveto(2,5); lineto(3,4);
     linerel(1,-1); linerel(1,0);  linerel(1,-1);
     linerel(1,0);  linerel(0,1);  linerel(1,1);
     linerel(1,1);  linerel(-1,1); linerel(-1,1);
     linerel(-1,1); linerel(-1,-1); linerel(-1,1);
     linerel(-1,-1); linerel(-1,0); linerel(1,-1);
      linerel(-1,-1);
    setcolor(1);  /* green */
     moveto(0,3); lineto(1,1);
      linerel(1,1);  linerel(1,-1);  linerel(0,-1);
      linerel(-1,-1); linerel(1,0);  linerel(1,1);
      linerel(1,0);   linerel(0,-1); linerel(1,-1);
      linerel(1,1);   linerel(1,0);  linerel(1,1);
      linerel(-1,1);  linerel(1,1);  linerel(1,1);
      linerel(-1,1);  linerel(0,1);  linerel(-1,1);
      linerel(-1,1);  linerel(0,1);  linerel(-1,-1);
      linerel(-1,-1); linerel(-1,1); linerel(-1,0);
      linerel(0,-1);  linerel(-1,0); linerel(-1,-1);
      linerel(1,-1);  linerel(-1,-1); linerel(0,-1);
      linerel(-1,-1);
      putpixel(7,8,3);  putpixel(8,7,3);
      putpixel(8,2,3);  putpixel(6,1,3);
      putpixel(2,3,3);  putpixel(1,2,3);
      putpixel(3,5,3);  putpixel(8,5,3);
  /* ** determine storage needed ** */
    numbytes = (unsigned int)imagesize(0,0,10,10);
  /* ** allocate buffer ** */
    burst = (char *)malloc(numbytes);
    getimage(0,0,10,10,burst); /* save the image */
     cleardevice();
  /* ** draw bomb figure ** */
   setcolor(2);   /* red */
   setlinestyle(0,0,1);
    putpixel(0,0,2); putpixel(2,0,2);
    moveto(0,2);  lineto(0,5);
    moveto(1,0);  lineto(1,6);
    moveto(2,2);  lineto(2,5);
  /* ** determine storage needed ** */
    numbytes = (unsigned int)imagesize(0,0,2,6);
  /* ** allocate buffer ** */
    bomb = (char *)malloc(numbytes);
    getimage(0,0,2,6,bomb);
     cleardevice();
}
```

freely. You'll be awfully glad you did.

So, let's look at what we have here for a game.

A Review of the TANK_WAR Action Game

As mentioned, there are five modules. At this point scoring is not included. That will come with the next in this series.

The TANK_WAR Module

This module, which contains **main()**, controls the action. The initial step of course is to set up the **graphics mode**. The second is the creation of the game's play objects. These are a fleet of bomber aircraft, organized in a flight group of three. There is an army tank, which is under control of the player. The player fires shells at the planes, so there has to be construction for these. And the planes drop bombs on the tank—a construction is in order here. When a shell or bomb makes its bang there is need for a burst. All of these are constructed in a dedicated module, GME_FIGS.C.

With the play objects all constructed and safely tucked away into buffers awaiting the call to action we can begin the screen activity. This commences with the **while()** loop beginning at line 42. The entire action of the game is controlled by this loop. The while is enabled by the integer "i" whose value is set to "1" in the declaration. To exit the game this variable is reset, that is, set to zero. Another key variable is **run**, which we will learn more of shortly.

As we take a look at the activity maintained by the loop it should become apparent why a loop is the only sensible approach to control of the game action. A point of importance is that only two actions are available to the player: control of the tank's direction and control of its gun firing. And so we have to look at the loop as the carrying out of two tightly related tasks: a continuation

of the screen action while enabling the player's input on the tank's operation.

In the normal development of a game of this kind operations are built up in layers as it were. First came the tank, followed by the bomber group. After working out the wrinkles, shell firing was added to the tank's activity. With the player able to fire shells means had to be developed to detect contact with any of the aircraft. A shell burst replacing both the shell and the target was designed. With sound. The bombers were taught when to drop the objects which would detonate on contact with the tank or the "ground." Once a bomb is dropped it has to be maintained, no matter what else goes on. All of this activity is reflected in the procedures embedded in the while loop.

As we review the other modules it will become very clear that the decision making is heavily dependent on the values taken on by flags. Flags are set and cleared as the action requires. The flag value becomes the basis for decisions on the kind of action to be taken. Right off in the loop we see this process in determining the direction the tank is to take in its trek across the screen. Three possibilities exist: move left, move right, or don't move at all. We will see at the end of the loop how the flag values are arrived at.

Which should raise the question of why the determination is made at the end of the loop. Why not at the beginning.

Okay. In order to get the game going in the first place some arbitrary decisions have to be made, one of them on the direction to be taken by the tank. The direction flag is **tdir_flg**, declared as a global integer with the initial value of 0. The tank will start off moving to the left. Not too illogical considering its initial coordinates, **int leftt_col = 200, ritet_col = 219;**, also a global, place it at the extreme lower right corner of our screen.

With the tank underway the program now must set the remainder of the agenda into motion. This it does with a call to **bomber()** (line 51). **bomber()** is located in another module, **BOMR.C**. Each call to **bomber()** must perform a routine of tasks. These we will look at shortly.

The first check is to ascertain there really is a bomber available. A non-existent bomber should not be dropping bombs. If any of the hit flags still read zero there really is one available. In this event a call is made to a function by name **drop_bombs()** (line 53, TANK_WAR.C). **drop_bombs()** is in its own module, **BOMBS.C**, Listing 7. Offhand dropping a bomb might appear to be a relatively simple task. Well, read on and see.

The call to **shel(run)** passes information on whether a shell has been fired, and if so at which angle. The shell firing keys are the A, S, D, and F. The test is performed at the **SHELL.C** module, Listing 8. If no key has been pressed the function returns. If a key has been pressed run is reset on the return.

Line 55 is a second call to **drop_bombs()**. The purpose here is to speed up the descent. Bombs are dropped incrementally unlike shells which streak off at their target without interruption.

At this time, line 56, the bombers are cleared. Recall that the call to display the bombers was made back in line 51. The intervening action takes place with these in view. With this call they are erased from the screen. The purpose of this scheme is to simply maintain the aircraft in view while it is being shot at and/or dropping a bomb.

At this time the tank is displayed, line 57. The call to **putimage()** is followed by a screen edge test. If the tank is at a limit its direction flag is reversed. Note that the remainder of the loop is leaped over in this event. This to prevent the player from counteracting the change before realizing it has taken place.

Listing 6. The bomber aircraft group for the TANK_WAR action game.

```
 1: /* BOMR.C
 2: ** Construction of a three plane, left-to-right aircraft group.
 3: ** A module of TANK_WAR.C
 4: ** Compiled with Turbo C version 2.0 graphics library.
 5: */
 6: #include <stdio.h>
 7: #include <graphics.h>
 8:   /* ** global declarations  ** */
 9: int leftbd_col = 10,leftbd_top = 20, riteb_col = 30;
10: int leftbe1_top, leftbe2_top, leftbe3_top;
11: int leftbe1_col, leftbe2_col, leftbe3_col;
12: int bh1_hit = 0, bh2_hit = 0, bh3_hit = 0;
13: extern char far *bomr;
14:   /* == Begin program == */
15: bomber()
16: {
17:   if(bh1_hit == 1 && bh2_hit == 1 && bh3_hit == 1) {
18:    leftbd_col += 100; riteb_col += 100;
19:     bh1_hit = 0; bh2_hit = 0; bh3_hit = 0;          }
20:   /* ** begin animation ** */
21:   /* ** draw first bomber ** */
22:     leftbe1_col = leftbd_col; leftbe1_top = leftbd_top;
23:      if(bh1_hit == 0) draw_bomber();
24:   /* ** draw second bomber ** */
25:     leftbd_col -= 10;  leftbd_top += 20; riteb_col -= 10;
26:     leftbe2_col = leftbd_col; leftbe2_top = leftbd_top;
27:       if(bh2_hit == 0) draw_bomber();
28:   /* ** draw third bomber ** */
29:     leftbd_col += 25;  leftbd_top += 5; riteb_col += 25;
30:     leftbe3_col = leftbd_col; leftbe3_top = leftbd_top;
31:      if(bh3_hit == 0) draw_bomber();
32:   /* ** advance to next position ** */
33:     riteb_col -= 5; leftbd_col -= 5; leftbd_top = 20;
34:     if(riteb_col >= 299)  {
35:       leftbd_col = 10, riteb_col = 30;
36:                           }
37: }
38:   /* == erase bombers == */
39: clr_bomrs()
40: {
41:   /* ** erase first bomber ** */
42:    if(bh1_hit == 0)
43:     putimage(leftbe1_col,leftbe1_top,bomr,XOR_PUT);
44:   /* ** erase second bomber ** */
45:    if(bh2_hit == 0)
46:       putimage(leftbe2_col,leftbe2_top,bomr,XOR_PUT);
47:   /* ** erase third bomber ** */
48:    if(bh3_hit == 0)
49:       putimage(leftbe3_col,leftbe3_top,bomr,XOR_PUT);
50: }
51:   /* == draw bomber as sequence of h/v lines == */
52: draw_bomber()
53: {
54:   putimage(leftbd_col,leftbd_top,bomr,COPY_PUT);
55:  }
```

The remainder of the loop is concerned with keyboard input. A challenge was how to interpret the closure of any key, ASCII or non-ASCII, without echo with a single key press. We can do this using the function **rd_nonasky()** because of the manner interrupt 16H reads the keyboard. The non-ASCII keys place a 0 in register AL and an integer for the key in AH. An ASCII key press places the ASCII value in AL.

The first requirement is to check the keyboard for a closure. If there is none waiting there is no point in going on to the end of the loop. If **kbhit()** has a value of zero the program returns to the beginning of the loop. If a non-zero value exists a check is made of the keypad arrow keys, 4,6, and 8. If one of these non-ASCII keys has been pressed its value is in variable n_asc. If this is not the case either a meaningless key or an ASCII key has been pressed. Assuming an ASCII key closure its char value is assumed by the variable run. Pressing the Q key exits the program.

Now that we have looked at how the program control performs let's make a quick review of the four supporting modules.

The Game Object Construction Module

The constructions in **GME_FIGS.C** all use **putimage()**. To date I have not found a way to combine **putimage()** with viewports in animation without the whole thing blowing up. Viewports do work with **putimage()** for static displays, such as the game scoring

Listing 7. The BOMBS module for the TANK_WAR action game

```
. 1: /* BOMBS.C
 2: ** The bombs module for TANK_WAR.C.
 3: ** Compiled with Turbo C version 2.0 graphics library.
 4: */
 5: #include <graphics.h>
 6:   /* ** global declarations ** */
 7: extern char far *bomb;
 8: extern char far *burst;
 9: extern char far *tank;
10: extern int lefft_col;
11: extern int leftbe1_col, leftbe2_col, leftbe3_col;
12: extern int bh1_hit, bh2_hit, bh3_hit;
13: int tnk_hit = 25;
14: int tnk_hit_flg = 0;
15:   /* == begin program == */
16: drop_bombs()
17: {
18: static int xa = 0, ya = 0;
19: static int drp1_flg = 0, drp2_flg = 0, drp3_flg = 0;
20: static int bom1_flg = 1, bom2_flg = 1, bom3_flg = 1;
21: static int xbmb1,xbmb2,xbmb3;
22: static int ybmb1 = 65,ybmb2 = 69,ybmb3 = 73;
23:   /* ** begin animation ** */
24:   /* ** bomber 1 bomb drop ** */
25:    if(bh1_hit == 0 && (lefft_col-leftbe1_col)
26:     <= 40 && bom1_flg == 1) {
27:      drp1_flg = 1;  bom1_flg = 0;
28:      xbmb1 = leftbe1_col + 6;      }
29:     if(drp1_flg == 1)        {
30:      bmb1_drp(xbmb1,ybmb1);
31:        ybmb1 += 6;             }
32:          if(ybmb1 >= 189 || (ybmb1 >= 189 && xbmb1
33:           >= lefft_col && xbmb1 <= lefft_col + 20)) {
34:          xa = xbmb1; ya = 189;
35:          bburst(xa,ya);  {
36:              drp1_flg = 0; bom1_flg = 1; ybmb1 = 65;
37:                                  }}
38:   /* ** bomber 2 bomb drop ** */
39:    if(bh2_hit == 0 && (lefft_col-leftbe2_col)
40:      <= 40 && bom2_flg == 1) {
41:       drp2_flg = 1; bom2_flg = 0;
42:      xbmb2 = leftbe2_col + 12;      }
43:     if(drp2_flg == 1)        {
44:       bmb2_drp(xbmb2,ybmb2);
45:        ybmb2 += 6;             }
46:          if(ybmb2 >= 189 || (ybmb2 >= 189 && xbmb2
47:           >= lefft_col && xbmb2 <= lefft_col + 20)) {
48:          xa = xbmb2; ya = 189;
49:          bburst(xa,ya);  {
50:              drp2_flg = 0; bom2_flg = 1; ybmb2 = 69;
51:                                  }}
52:   /* ** bomber 3 bomb drop ** */
53:    if(bh3_hit == 0 && (lefft_col-leftbe3_col)
54:       <= 40 && bom3_flg == 1) {
55:       drp3_flg = 1;  bom3_flg = 0;
56:       xbmb3 = leftbe3_col+6;                          }
57:      if(drp3_flg == 1)       {
58:        bmb3_drp(xbmb3,ybmb3);
59:         ybmb3 += 6;             }
60:           if(ybmb3 >= 189 || (ybmb3 >= 189 && xbmb3
61:            >= lefft_col && xbmb3 <= lefft_col + 20)) {
62:           xa = xbmb3; ya = 189;
63:           bburst(xa,ya);  {
64:               drp3_flg = 0; bom3_flg = 1; ybmb3 = 73;
65:                                  }}
66: }
67: /* ** display shell burst ** */
68: bburst(short xa,short ya)
69: {
70:     if(xa >= lefft_col && xa <= lefft_col + 20)
71:     putimage(lefft_col,189,tank,XOR_PUT);
72:     putimage(xa,ya,burst,COPY_PUT);
73:   /* ** ring bell ** */
74:     sound(1000);
75:     delay(50);
76:     sound(1500);
77:     delay(50);
78:     sound(2000);
79:     delay(50);
80:     sound(1000);
81:     delay(50);
82:     nosound();
83:   putimage(xa,ya,burst,XOR_PUT);
84:   if(xa >= lefft_col && xa <= lefft_col + 20) {
85:   putimage(lefft_col,189,tank,COPY_PUT);
86:   /* ** update tank hit scoring ** */
87:    tnk_hit_flg = 1;  tnk_hit -= 1;              }
88: }
89:   /* == draw descending bomb 1 figure == */
90: bmb1_drp(int xbmb,int ybmb)
91: {
92:   putimage(xbmb,ybmb,bomb,COPY_PUT);
93:      delay(15);
94:   putimage(xbmb,ybmb,bomb,XOR_PUT);
95: }
96:   /* == draw descending bomb 2 figure == */
97: bmb2_drp(int xbmb,int ybmb)
98: {
99:    putimage(xbmb,ybmb,bomb,COPY_PUT);
100:      delay(15);
101:   putimage(xbmb,ybmb,bomb,XOR_PUT);
102: }
103:   /* == draw descending bomb 3 figure == */
104: bmb3_drp(int xbmb,int ybmb)
105: {
106:   putimage(xbmb,ybmb,bomb,COPY_PUT);
107:      delay(15);
108:   putimage(xbmb,ybmb,bomb,XOR_PUT);
109: }
```

to be developed in the next of the series. But even with these problems can develop.

The library graphics do have a bad feature in that the construction of each object is accompanied by a brief flash of its image at the screen's HOME position. To date I have not found a way around this. Hopefully someday.

Five play objects are created in this module: the tank, a bomber, the shell fired from the tank, the bomb dropped from a plane, and the burst resulting from an exploding shell or bomb. They are declared as far pointers in lines 8 - 12. The declarations must be global. All the modules are created in the one function, **drw_figs()**. Two declarations are required: a buffer to store the image temporarily, and an unsigned variable to hold the imagesize. All of the play objects are constructed from straight line segments. After the object's design is specified the storage need is determined, the buffer allocated, and the image stored using a call to **getimage()**. This is followed by a brief flash on clearing the screen.

The BOMR() Module

When we look at Listing 6, we see that lines 20 - 31 have responsibility for displaying each of the three planes. To get to this point, however, a preliminary screening process takes place in lines 17 -19. Again a reliance on flags (BH stands for Bomber Hit). The AND (&&) logic reports if all the bombers have been shot down. If that is so, they must be re-constructed. New screen coordinates are assigned as offsets in a brazen attempt to confuse the player on just where it is they may re-appear.

If we have shot down a bomber we jolly well do not want it to continue on its trek across our screen. We want it to go away! Right? So if the flag is set to 0 the plane appears. When a plane receives a shell hit, as we shall shortly learn, its hit flag is set to 1. In which case it is not drawn.

So why are its column coordinates updated as though it were really present? So the program will always know where it should be, even if it is not visible, that's why. Just a simple matter of keeping track for its later re-appearance. Bomber drawing is performed with **putimage()** in its COPY_PUT action constant mode.

After our bombers have been drawn and been shot at, assuming they were not shot down, they have to be advanced to the next location on screen. It is a good idea to erase the existing image prior to displaying another at its new location. The function **clr_bomrs()** attends to that. Erasing is performed with **putimage()** in its XOR_PUT action constant mode.

The BOMBS Module

When it comes to dropping bombs there are a lot of flags and other variables to keep track of. When you think about it, bombs do require some conservation. We shouldn't just sprinkle them

Listing 8. The SHELL FIRING module for the TANK_WAR action game.

```
 1: /* SHELL.C
 2: ** Program to animate an artillary shell in any one
 3: ** of four angles: 50, 70, 110, or 130 degrees.
 4: ** A module of TANK_WAR screen action game.
 5: ** Compiled with Turbo C version 2.0 graphics library.
 6: */
 7: #include <stdio.h>
 8: #include <graphics.h>
 9: void sburst(short x1,short y1);
10:   /* ** global declarations ** */
11: extern int lefft_col, ritet_col;
12: extern int bh1_hit, bh2_hit, bh3_hit;
13: extern int leftbe1_col, leftbe1_top;
14: extern int leftbe2_col, leftbe2_top;
15: extern int leftbe3_col, leftbe3_top;
16: extern int drp1_flg, drp2_flg, drp3_flg;
17: extern int xbmb1,xbmb2,xbmb3,ybmb1,ybmb2,ybmb3;
18: extern char far *shell;
19: extern char far *burst;
20: extern char far *bomr;
21: int score = 0;
22: int pla_hit = 0;
23: int flite_flg = 0;
24: int shl_bal = 150;
25: int shl_fir_flg = 1;
26:   /* == Begin program == */
27: shel(char tnk_shl)
28: {
29:   if(tnk_shl != 'A' && tnk_shl != 'S'
30:     && tnk_shl != 'D' && tnk_shl != 'F') return;
31:   /* ** begin animation ** */
32:         dis_pla_shl(tnk_shl);
33: }
34:   /* == display shell burst == */
35: void sburst(short xa,short ya)
36: {
37:   putimage(xa-5,ya-5,burst,COPY_PUT);
38:    /* ** ring bell ** */
39:    sound(1000);
40:    delay(50);
41:    sound(1500);
42:    delay(50);
43:    sound(2000);
44:    delay(50);
45:    sound(1000);
46:    delay(50);
47:    nosound();
48:     putimage(xa-5,ya-5,burst,XOR_PUT);
49: }
50:   /* == create basic shell == */
51: tshell(int xb, int yb)
52: {
53:  int xbs = xb, ybs = yb;
54:   /* ** display and erase shell ** */
55:     putimage(xb,yb,shell,COPY_PUT);
56:      hit_tst(xbs,ybs);
57:      delay(8);
58:       putimage(xb,yb,shell,XOR_PUT);
59: }
60:   /* == test for aircraft hits == */
61: hit_tst(int xb,int yb)
62: {
63:   /* ** test for hit on bomber 1 ** */
64:    if(bh1_hit == 0) {
65:     if(xb >= leftbe1_col && xb <= leftbe1_col + 20
66:        && yb >= leftbe1_top && yb <= leftbe1_top + 20)
67:                   {
68:        sburst(xb+10,yb-10);
69:       putimage(leftbe1_col,leftbe1_top,bomr,COPY_PUT);
70:       putimage(leftbe1_col,leftbe1_top,bomr,XOR_PUT);
71:     putimage(xb,yb,shell,COPY_PUT);
72:         flite_flg = 0;
73:         bh1_hit = 1; pla_hit += 1; score += 100;
74:          return;    }}
75:   /* ** test for hit on bomber 2 ** */
76:    if(bh2_hit == 0) {
77:     if(xb >= leftbe2_col && xb <= leftbe2_col + 20
78:        && yb >= leftbe2_top && yb <= leftbe2_top + 20)
79:                   {
80:        sburst(xb+10,yb-10);
81:       putimage(leftbe2_col,leftbe2_top,bomr,COPY_PUT);
82:       putimage(leftbe2_col,leftbe2_top,bomr,XOR_PUT);
83:     putimage(xb,yb,shell,COPY_PUT);
84:         flite_flg = 0;
85:          bh2_hit = 1;pla_hit +=  1; score += 100;
86:          return;    }}
87:   /* ** test for hit on bomber 3 ** */
88:    if(bh3_hit == 0) {
89:     if(xb >= leftbe3_col && xb <= leftbe3_col + 20
90:        && yb >= leftbe3_top && yb <= leftbe3_top + 20)
91:                   {
92:        sburst(xb+10,yb-10);
93:       putimage(leftbe3_col,leftbe3_top,bomr,COPY_PUT);
94:       putimage(leftbe3_col,leftbe3_top,bomr,XOR_PUT);
95:     putimage(xb,yb,shell,COPY_PUT);
96:         flite_flg = 0;
97:         bh3_hit = 1; pla_hit += 1; score += 100;
98:          return;    }}
99: }
100:   /* == read key and fire shell == */
101: dis_pla_shl(char shl_ang)
102: {
103: int x = lefft_col + 10, y = 179, ang_flg;
104:  flite_flg = 1;  shl_fir_flg = 1; shl_bal -= 1;
105:   switch(shl_ang)  {
106:      case 'A': { ang_flg = 1; break; }
107:      case 'S': { ang_flg = 2; break; }
108:      case 'D': { ang_flg = 4; break; }
109:      case 'F': { ang_flg = 5; break; }
110:       default: return;
111:                }
112:   while(flite_flg) {
113:    if(ang_flg == 1) { x -= 4; y -=5; tshell(x,y);
114:     if(flite_flg == 0) break; }
115:    else if(ang_flg == 2) { x -= 3; y -=5; tshell(x,y);
116:     if(flite_flg == 0) break; }
117:    else if(ang_flg == 4) { x += 3; y -=5; tshell(x,y);
118:     if(flite_flg == 0) break; }
119:    else if(ang_flg == 5) { x += 4; y -=5; tshell(x,y);
120:     if(flite_flg == 0) break; }
121:      if(y <= 22) { flite_flg = 0; return; }
122:                }
123: }
```

like wildfires across the screen in hopes of knocking out a tank. So it's back to decision making logic again. The variable declarations are all static as their values must be maintained between function calls.

Bombs descend in steps of six pixel increments. We have first of all to make sure there is a plane available. Then its screen position relative to its target, the tank, has to be checked. If these tests are positive and a bomb flag, bomi_flg, is enabled, where the i is 1, 2, or 3 depending on the aircraft, a drop flag, drpi_flg, is enabled. At this time the bomb flag is cleared, else next time around a new bomb may be dropped, ready or not. With each call to **drop_bomb()** the bomb makes a descent until it strikes the tank or the "ground." At that point the burst function is called with the burst coordinates and the flags are set up for the next drop.

There is a complication with the screen images when one intersects another, such as when a bomb or shell strikes an object. First we want the shell to go away. Then we want the burst to appear overlaying the object underneath, if any. Then, if there has been a hit on an object it has to go away. Well, the XOR_PUT action constant for a given object does not necessarily affect another. Without some cleanup provisions our screen is quickly littered with debris, sort of like the space junk now orbiting the earth. So as part of the cleanup procedure lines 92 and 94 clean up the debris from bomber 1 by a quick draw and erasure. Even so there will be an occasional display of lingering garbage, which is swept away by the next pass across the screen.

The SHELL Module

By now we have a pretty fair understanding of all the concerns to be considered in moving screen objects around. Even so the large number of global variables in this module may be a surprise. This simply because there is so much to keep track of. First we have to know where the tank is when the shell is fired. Shells mysteriously appearing from out of nowhere are not uncommon in working out some of these coordinate schemes. And as the little guy goes streaking off into the sky its motions require continuous draw, delay, clear, advance instructions.

The actual shell firing takes place in the last function, **dis_pla_shl(char shl_ang)**, beginning at line 101. The only flag

actually used at this time is the flite_flg which maintains the shell in a transit loop. The shl_fir_flg and variable shl_bal are used for scoring. This istrue for the variables pla_hit and score as well. The scoring of hits and keeping track of shells fired is actually relatively simple. The reason for not including them at this time is that their display on the screen is not.

The shell may or may not hit one of the bombers. So the relation of each bomber to the shell must be maintained. With each shell advance detection logic must test for an intersection. The testing is performed in the function **hit_tst()**, beginning on line 61. As we saw in the **BOMBS** module the **putimage()** action constants COPY_PUT and XOR_PUT perform debris cleanup.

It may come as a surprise that a burst function is included in two modules. This is largely as a matter of convenience for passing location variables.

Summary

We have learned that bringing all the play object action together on our screen does involve a fair amount of complexity. Since activities of this kind require a lot of program code it is to our advantage to write individual modules and link them together. The Turbo C MAKE utility is of considerable assistance in the compiling and linking operations.

This article of necessity starts right out with the completed modules.

In a real development each play object is written in a stand-alone mode to verify its construction. When we are satisfied with its performance it is revised for linking in with the overall program. The interactive logic must be developed with the entire system, but doing what can be done in the stand-alone mode eases this task. ●

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**
Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**
The renowned Z-System command processor (ZCPR v 3.4) and companion utilities. Dynamically change memory use. Installs automatically
Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

ZMATE **$50**
New Z-System version of renowned PMATE macro editor with split-screen mode for two-window viewing of one or more files. Extremely powerful and versatile macro capability lets you automate repetitive or complex editing tasks, making it the ultimate programmer's editor. Macros can be saved for reuse and also assigned to keys. Editing keys can be reconfigured for personal style. Supports drive/user and named-directory file references. Auto-installs on Z systems. Z-80 only. Supplied with user manual and sample macro files.

PluPerfect Writer **$35**
Powerful text and program editor with EMACS-style features. Edit files up to 200K. Use up to 8 files at one time, with split-screen view. Short, text-oriented commands for fast touch-typing: move and delete by character, word, sentence, paragraph, plus rapid insert/delete/copy and search. Built-in file directory, disk change, space on disk. New release of our original upgrade to Perfect Writer 1.20, now for all Z80 computers. On-disk documentation only.

ZSDOS ..... **$75, for ZRDOS users just $60**
State-of-the-art operating system. Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**
Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**
Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**
Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402
(213)-393-6105 (eves.)

BackGrounder ii ©, DosDisk ©, Z3PLUS ©, PluPerfect Writer ©, JetFind © Copyright 1986-88 by Bridger Mitchell.

Real Computing

The National Semiconductor NS320XX

by Richard Rodman

Metal 0.6

Version 0.6 of Bare Metal is available now. It has been completely transformed through the use of a new assembler by Bruce Culbertson. It is now linked together from separately-assembled or compiled modules. Phil Prendeville's C compiler has been modified for compatibility with the assembler.

Metal is now being distributed as a complete cross development package for a PC. For example, if you want to put the system in a directory C:\NS32, put the floppy in the A: drive, and enter the command:

```
install c: ns32
```

Notice the space between the C: and the NS32. MS-DOS has such a stupid command interpreter that the drive and the directory have to be passed as separate items.

At any rate, along with the compiler and assembler you also get a *make* utility, a new version of the *host* program, an EPROM utility, and other miscellaneous utilities. Naturally, source code is provided for everything. BIOSes are provided for the Cromemco 16FDC and for the PD-32 coprocessor board.

The new version of *host* supports the TDS download format. (The previous one claimed to, but testing showed that it didn't.) The simple ROM monitor, SRM, has been modified to allow TDS download format and commands.

Work is under way to supply a complete C library and include files. This will allow the compiler and assembler to run in *native* mode, and a PC will no longer be necessary.

There are a number of major internal changes in version 0.6. In order to get around the address-constant problem, we've basically given up and created a new kind of executable that executes at a CP/M-like TPA address of 010000 hex (64K). These files should be linked as follows (for a hypothetical program *myprog*):

```
ld -e __main -T 010000 -o myprog.e myprog.o -lc
```

Not very user-friendly, you say? All suggestions and help are welcome!

Executables generated in the .E32 format, which are completely relocatable, are still supported. Hopefully, in the future, an approach to solving the address constant problem can be determined that will not require the TPA restriction; unless you have an MMU (not currently supported), only one program can be running at a time, like CP/M.

Another internal change is that the single-character I/O system calls are no longer supported. They just turned out to be too inefficient. New system calls for the *seek* and *tell* functions (get and set read/write position in a file) have been added.

Rules for 360s and 1.2s

There are two simple rules for exchanging floppies between 360 KB and 1.2 MB floppy drives. These are as follows:

1. If you format it on the 360, you can write it on the 360, but if you ever write it on the 1.2, you can't read it on the 360 anymore.
2. If the floppy has *never, ever* been formatted on any drive, and you format it /4 on the 1.2, you can write it on the 1.2; but if you ever write it on the 360, and *then* write it on the 1.2, you can't read it on the 360 anymore. Reformatting on the 1.2 won't help; if you reformat on the 360, see rule number 1.

Remember that any writing to the disk involves writes to the directory, too, so don't imagine that "accessing different files" will keep you safe.

Maybe it's about time we say to heck with the silly nonsense and standardize on 3.5 inch floppies!

Speaking of standards, I recently saw a Zenith MinisPort, and was pleasantly surprised. The keystroke feel was nice, the display was quite easy to read (it's adjustable to any angle), the 2-inch floppy is kind of spiffy, and the thing is amazingly light and small! Why are all the reviewers so intent at bad-mouthing it? Because it has a lowly 8088 processor, I suppose. Tests with laboratory animals have proved that using a faster CPU actually increases typing speed.

ICU test comments

The ICU test program presented in a previous column will not work on the PD-32 board. It seems that the H-counter is used on the PD-32 board to control dynamic memory refresh. The general-purpose I/O bits of the ICU are used to control various other functions of the board, so be careful.

The CD conspiracy

There are two shocking facts about CDs that are being kept secret by a mammoth conspiracy. Tell everyone you know!

First, CDs are by no means indestructible. The CD contains a very thin top layer (the printed side), which is *very* delicate. So delicate, in fact, that thumbnails pressing on, or even small objects pressing on or dropped on this side will destroy the disk.

Worse, however, is the Notch. It seems that some people in the entertainment industry wanted a copy-protection scheme for the DAT (Digital Audio Tape) in which there would be a 45 dB notch at 15 kHz. Essentially, frequencies in that region would be completely wiped out, which (as any audio buff will tell you) will cause distortions throughout the rest of the spectrum.

As with all copy-protection schemes, this idea punishes all legitimate users without really impacting any illegitimate ones; but the entertainment industry is a more monopolistic and openly oppressive cartel than the computer industry has ever had. They tried to get Congress to force this on everyone, but fortunately this attempt failed.

But here's the sickening part: The electronics industry is doing it *anyway!* Not only do all DAT decks incorporate the Notch, but more and more CDs are being produced and distributed recorded

with the Notch.

A Real Computer

Dave Rand, the designer of the PD-32 board, has designed a new board which is considerably more powerful. It has a 32532 CPU equipped with up to 32 megabytes of no-wait-state, page-access SIMM DRAM. It has 8 serial ports. It has two SCSI ports, one for a disk and/or tape, and one for other uses. It also can be equipped with a 32381 math coprocessor. The system is offered as an unpopulated four-layer printed circuit board for $200. If you're interested, contact him at the address below.

Bruce Culbertson and others are working on porting Minix to the board, and are reported to be close to finished. He is working on an add-on board to include an Ethernet port and 16 additional serial ports.

The design of this board really gets around a lot of the limitations of the Designer's Kit board, mainly limited RAM and I/O. After all, what good is a fast processor if you can't get data in and out of it?

What, after all, makes a system fast? We've all seen 8080 systems that offer lightning response—and 386 systems that move like glaciers. For a really fast system, all components have to be tuned together, including the software.

State machines

Tuning implies music, and music implies rhythm. In an embedded system, it often doesn't matter so much how fast you do something, as how close to the right time you do something; so, embedded systems are often designed around a synchronous, tick-derived state machine. Such a design yields a deterministic—accurate and predictable—response.

However, a synchronous state machine does not lend itself to dynamic, constantly-changing systems such as general-purpose computers. Many modern multitasking kernels offer a hybrid approach, where tasks can be implemented as synchronous state machines or as dynamic tasks.

It has been considered a truism that real-time operating systems make poor development environments. Real-time operating systems cannot allow priorities to be changed; they can't allow vital tasks to be swapped; they can't use high-overhead, dynamic-block, multiply-indirect file systems. However, I think as the kernel design advances, as processors get better, and as the low-overhead system design approach catches on, we'll see systems with all of the friendliness of Unix (or more, hopefully) that really are suitable target environments for real-time systems. What would be really great is if you had binary compatibility from the bottom of the line to the top, like the VAX, but in a micro.

Next Time

Futurebus: Is it going places, or taking people for a ride? •

Where to Call or Write

BBS: 1-703-330-9049

Dave Rand, 1-408-733-4125

User Disk

The complete installation package of Metal 0.6, including C compiler, assembler, linker, make utility, ROM monitor, host program, and various utilities, is available on 3.5" 1.44MB, 3.5" 720K, or 5.25" 360K PC format disks for $13 postpaid in the U.S.

Editor

(Continued from page 3)

You'll hear more about these products here, but contact them for more information in the meantime.

Ampro Little Board Returns

The Ampro Z80 Little Board which had been out of production has been picked up by Davidge Corporation, 94 Commerce Drive, Buellton, CA 93427, Phone (805) 688-9598. Dean Davidge has been producing his own line of single board computers for quite a while. I had one of his Z80 boards and it was a fine product, so he should do a good job with the Ampro LB.

Davidge is providing repair service for units made by both Ampro and Davidge. They can also provide the Project Board/80, CP/M and ZCPR3, BIOS Source, and manuals. As far as I know, this is one of the last CP/M computers still in production—support them if you want a source to be available in the future.

Another Davidge product line is the RIM (Remote Interface Multiplexer) Bus series of data collection and control cards.

Now that the Ampro LB is again available, I would like to develop a ROM based system to use the card as an embedded stand alone controller. This will be for controller applications, not desk top computer applications, and will be very different from the usual CCP, BIOS, and BDOS. Any one else interested in this? •

Cross-Assemblers as low as $50.00
Simulators as low as $100.00
Cross-Disassemblers as low as $100.00
Developer Packages
as low as $200.00(a $50.00 Savings)

A New Project
Our line of macro Cross-assemblers are easy to use and full featured, including conditional assembly and unlimited include files.

Get It To Market--FAST
Don't wait until the hardware is finished to debug your software. Our Simulators can test your program logic before the hardware is built.

No Source!
A minor glitch has shown up in the firmware, and you can't find the original source program. Our line of disassemblers can help you re-create the original assembly language source.

Set To Go
Buy our developer package and the next time your boss says "Get to work.", you'll be ready for anything.

Quality Solutions
PseudoCorp has been providing quality solutions for microprocessor problems since 1985.

BROAD RANGE OF SUPPORT

- Currently we support the following microprocessor families (with more in development):

Intel 8048	RCA 1802,05	Intel 8051	Intel 8096
Motorola 6800	Motorola 6801	Motorola 68HC11	Motorola 6805
Hitachi 6301	Motorola 6809	MOS Tech 6502	WDC 65C02
Rockwell 65C02	Intel 8080,85	Zilog Z80	NSC 800
Hitachi HD64180	Motorola 68000,8	Motorola 68010	Intel 80C196

- All products require an IBM PC or compatible.

So What Are You Waiting For? Call us:
PseudoCorp
Professional Development Products Group
716 Thimble Shoals Blvd, Suite E
Newport News, VA 23606
(804) 873-1947 FAX: (804)873-2154

(Continued from page 4)

it's somewhat **painful to interface** with industrial-class devices, such as milling machines, motors, sensors, etc.; and floppies don't work well in oily, smoky, electrically noisy or corrosive environments. However, I believe the PC **sets expectations** as to price and performance. "Why should I buy a $1000 single-board computer when I can get an AT for that price?": never mind that the AT couldn't do the job half as well. And can **you** build a floppy controller for $19, or a video card for $25? This trend will only get worse as new XTs drop in price, and used XT are pushed out of industrial use by cheap AT-class machines (or RISC machines as Lee suggests).

Actually, my more interesting argument is not that IBM-type PC's are cheap **end-use** systems; they are cheap **development** systems. There are good reasons that chip manufacturers sell their development tools to run on PCs. Typically, a PC-compatible processor development package includes a plug-in card, assembler, debugger and docs for several hundred dollars and up, with $1000 a common price. PC-class cross-compilers for C (and occasionally other languages) are available for a few to several hundred dollars and up. There are even shareware sources for a few cross-compilers, with registrations under $100. And there are more and cheaper tools for **native development** of 8088-class applications.

However, I have to admit that running a C cross compiler on a PC to cram code into the 4K program space of an 8051 which will operate a voice-responding home security system is a bit of overkill. And, as it turns out, a lot of work to convince the compiler not to use the "huge" model of 1 gigabyte addressing to make the code fit at all! The engineer or programmer becomes very removed from the application when you are designing a "talking doorknocker" from a screen/disk/keyboard environment. And for "high-powered" applications with multiple processors in constant communication, the generic cross-compiler/simulator is not very useful: the development system usually **is** the application system.

I think the answer is a **modular development environment**, where you assemble physical and software resources as needed. It could be as small as a few modules on your desk and a calculator-like interface; or as large as a PC connected to several processors. Lee Hart may have something with his "Z80 on a shingle" product and his "spreadsheet" programming metaphor. Now, if I can only get him out of the "2K teensie-weensie BASIC" world of development systems, we might get somewhere!

I have more thoughts on the subject, if your readers are interested, but I have to get off my PC-AT and go into the basement to check my memory test run on some customer's S-100 IMSAI 32K DRAM boards. The "heat test" hairdrier sometimes slips off and starts cooking my 8-inch floppies!

Herb Johnson

Forth Advocate

I am impressed with your publication to the extent that I must congratulate the staff for a job well done! The contributors, both regular and otherwise, usually accomplish their duty of generating fascinating and useful articles. Moreover, I feel compelled to contribute as well. I see you would like more contributors and am curious about your format preferences. Do you have any? I have such documents for various publications and would rely on them for any assistance otherwise. EDN, Personal Engineering, and Micro Cornucopia are the others I speak of, and as you may well know they all have different "flavors."

As long as I mentioned Micro Cornucopia, I must say I do not wish TCJ to become *The Last of the Mohicans* amongst the various magazines still oriented towards serious fringe computing. That is why I must contribute along with a general support of the advertisers, etc. You WILL receive a manuscript or two from me. I just need to get up off my rear to do it. I'm quite sure most other TCJ readers have other things to do, besides document their personal projects. Although, after going through the work of a project, documentation and publication really is worth it. It's a great way to make contacts and gain a reputation. Not to mention the satisfaction gained by the editorial approval.

As far as suggestions go, I would like a Neural Network article that is practical. I've contacted Dave Weinstein about his OOF (Object Oriented Forth) efforts and thought that OOF would be a good way of going about it. You may well agree. I am glad that Forth is a mainstay of TCJ. You can bet that I always hope Forth comes in every issue I receive! Also, the slant towards microcontrollers lately has been welcome. My main gripe about most articles in *Embedded Systems Programming* is that they are not very practical or useful. Pseudo code may be a good thing to some, but to me it seems quite useless. I like articles with schematics and real code, not fairy tails. Another thing I have been interested in lately is the CEBus development and articles in Circuit Cellar INK about them. Maybe that would be an opportunity for some people to contribute.

Anyways, I won't waste any more of your paper. Keep up the excellence and thanks.

Ronn Guttmann
Australia

Editor's Note: Our Forth authors are all overloaded with work, so there are no Forth articles in this issue—but there are more coming.

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. **Please allow six weeks notice. Thanks.**

THE COMPUTER JOURNAL

Back Issues

Special Close Out Sale
on these back issues only
Issues—1, 2, 3, 4 and 8
3 or more, $1.50 each postpaid in the U.S.
Outside of the U.S., 3 or more, $2.50 each postpaid surface.
Other back issues are available at the regular price.

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 8:

- **Build VIC-20 EPROM Programmer.**
- **Multi-User: CP/Net.**
- **Build High Resolution S-100 Graphics Board: Part 3.**
- **System Integration, Part 3: CP/M 3.0.**
- **Linear Optimization with Micros.**

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System.
- Database: A continuation of the data base primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking executive.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.
- The Computer Corner

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8088 software to produce modifiable assem. source code.
- Real Computing: The NS32032.
- S-100: EPROM Burner project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System. Part 1: Selecting your assembler, linker and debugger.
- The Computer Corner

Issue Number 36:

- Information Engineering: Introduction.
- Modula-2: A list of reference books.
- Temperature Measurement & Control: Agricultural computer application.
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE.
- Real Computing: NS32032 hardware for experimenter, CPUs in series, software options.
- SPRINT: A review.
- REL-Style Assembly Language for CP/M & ZSystems, part 2.
- Advanced CP/M: Environmental programming.
- The Computer Corner.

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers.
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER.
- Information Engineering: Basic Concepts: fields, field definition, client worksheets.
- Shells: Using ZCPR3 named shell variables to store date variables.
- Resident Programs: A detailed look at TSRs & how they can lead to chaos.
- Advanced CP/M: Raw and cooked console I/O.
- Real Computing: The NS 32000.
- ZSDOS: Anatomy of an Operating System: Part 1.
- The Computer Corner.

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS: Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

Issue Number 42:

- Dynamic Memory Allocation: Allocating memory at runtime with examples in Forth.
- Using BYE with NZCOM.
- C and the MS-DOS Screen Character Attributes.
- Forth Column: Lists and object oriented Forth.
- The Z-System Corner: Genie, BDS Z and Z-System Fundamentals.
- 68705 Embedded Controller Application: An example of a single-chip microcontroller application.
- Advanced CP/M: PluPerfect Writer and using BDS C with REL files.
- Real Computing: The NS 32000.
- The Computer Corner

Issue Number 43:

- Standardize Your Floppy Disk Drives.
- A New History Shell for ZSystem.
- Heath's HDOS, Then and Now.
- The ZSystem Corner: Software update service, and customizing NZCOM.
- Graphics Programming With C: Graphics routines for the IBM PC, and the Turbo C graphics library.
- Lazy Evaluation: End the evaluation as soon as the result is known.
- S-100: There's still life in the old bus.
- Advanced CP/M: Passing parameters, and complex error recovery.
- Real Computing: The NS32000.
- The Computer Corner.

Issue Number 44:

- Animation with Turbo C Part 1: The Basic Tools.
- Multitasking in Forth: New Micros F68FC11 and Max Forth.
- Mysteries of PC Floppy Disks Revealed: FM, MFM, and the twisted cable.
- DosDisk: MS-DOS disk format emulator for CP/M.
- Advanced CP/M: ZMATE and using lookup and dispatch for passing parameters.
- Real Computing: The NS32000.
- Forth Column: Handling Strings.
- Z-System Corner: MEX and telecommunications.
- The Computer Corner

Subscriptions

	U.S.	Foreign (Surface) Total
1year (6 issues)	$18.00	$24.00
2 years (12 issues)	$32.00	$46.00 _____
Air Mail rates on request.		

Back Issues

16 thru #43	$3.50 ea.	$4.50 ea.
6 or more	$3.00 ea.	$4.00 ea.
#44 and up	$4.50 ea.	$5.50 ea.
6 or more	$4.00 ea.	$5.00 ea. _____

Issue #s ordered_____

Subscription Total _____

Back Issues Total _____

Total Enclosed _____

All funds must be in U.S. dollars on a U.S. bank

Name_____

Address_____

_____

_____

☐ Check ☐ VISA ☐ MasterCard Exp. Date____

Card #_____

Signature_____

The Computer Journal
190 Sullivan Crossroad, Columbia Falls, MT 59912
Phone (406) 257-9119 Mountain Time Zone

need to be in place and followed. We have a big problem now with the use of "C". Many organizations have gone to "C" as the solve all language. They feel by using "C" there is no need to institute all the standards I talk about. The truth is just the opposite. Unreadable and undocumented "C" code is just as possible as any other type of coding. You need standards for your own programmers and any outside consultants. We have treated software as a magical art, but in fact it is a simple science like many others. When we buy a car, you know you can go an buy a shop manual for it. That shop manual shows you how things go together, are repaired, schematics for trouble shooting, and basically the scientific rules by which the car was created. A software product should have nothing less than a "shop manual" as well.

EXORMACS

A few last comments on the Motorola Exormacs assembler. There are two items about that assembler which are important for our readers. The last ten years of work was done using the Exormacs assembler. I have gone to the Avocet assembler on the PC to replace the use of the Exormacs. In doing so I have discovered many things. The most important is how poor the Exormacs is as an assembler.

Some of the modules that assemble properly on the Exormacs will produce several hundred errors. At first I thought it was just minor differences between the Exormacs and the Avocet. Instead I have learned that the difference is due to limited error checking on the part of the Exormacs. I am now a confirmed believer that your assembler should have good tight error checking. As part of the absence of personal controls, the past programmer exploited all the loose structures the Exormacs assembler allowed them to use.

A recent case in point has to do with DC statements. DC statements allocate a place in memory and set what is stored there to a defined value. A DS statement however just sets the space aside, typically setting it at zero value. When a table of variable values is desired you can include that table by using the DC statements and the corresponding desired value. If you want tables in which an offset or pointer into the table is used instead of the exact address you typically will use DS statements to define the pointers into that table.

Those are the rules which the Exormacs manual states. If an "offset" value is placed at the beginning of the DS definitions, then the assembler will equate pointers for those defined items. Later you can use the defined items as pointers or offsets that are added to a register to fetch or change a value stored in that location. The assembler however should error out if a DC statement is given an "offset" value. The problem is an offset forces addressing to start at zero location, and as such you are directing the assembler to put specific values at those locations.

Well it turns out the Exormacs allows DC statements with "offsets" of zero. This is completely opposite of what the manual states. What actually happens is the assembler treats the DCs as if they were DSs. It returns pointers if we give the DCs a zero offset, but addresses if no offset is used. When the same code is used on the Avocet crossassembler hundreds of errors get produced because it adheres to the rules. It forces you to create lists of offset pointers different from the list of values to be stored there. For me it has made things more difficult, because not only must I now make two lists of pointers, but I have to seek out these differences wherever they maybe in 4 megabytes of source code. It really is a management of information problems. It makes it hard to find out which module is actually defining the space and which modules are just using the same list of DCs for offset pointers (they all use the same include file). With two separate lists, I can see the users from the source quickly and easily.

The biggest concern is the lack of error checking. If use of DCs with an "offset" is an error, say so. I have since found many cases where the assembler will just put what it thinks should go there instead of erroring out. What it thinks should go there and what you need to go there to make it work, may in fact be two different things. If I say do this, and that is not correct, I want to know! Without good error checking you may find yourself with bugs that are assembler generated, and can be solved only by disassembling the produced code. I have spent many hours of late doing just that. What I have discovered is a lot of places in which the code is not what was assumed or desired.

Because the managers of the project had so little control over the programmers, nobody has ever checked to see what the assembler produced. Their solution to bugs was adding more code to make up for the unknown source of the bug (the assembler we now know). Based on the errors I have seen on the Avocet, I guess there may be as many as 1000 lines of code improperly assembled that exist in the 50,000 lines of source code. To make it worse, I may have to find every one of those errors, before it can be assembled on a new assembler. No simple task.

Till Later

We spent this whole time on programming, but I think it is important. If your business is based on a software product, I hope you may look a little more seriously into what is happening within those begin...end statements. Next time I hope to talk about the Harris RTX2001A. ●

CP/M SOFTWARE
Including Amstrad

100 Page Public Domain Catalog,	$8.50 + $1.50 S&H
New Digital Research CP/M 2.2 manual	$19.95 + $3.00 S&H
Turbo Pascal	$65.00 + $2.50 S&H
Uniform	$65.00 + $2.50 S&H
LocaScript 2	$60.00 + $2.50 S&H
Nevada COBOL	$39.95 + $2.50 S&H
Out-Think	$69.95 + $2.50 S&H
Checks & Balances	$74.95 + $2.50 S&H
T/Maker	$120.00 + $2.50 S&H
SuperCalc	$99.00 + $2.50 S&H
EZDT (a Z80 DDT)	$17.95 + $2.50 S&H

Plus Much More

Over 400 Public Domain disks: Languages, Databases, Word Processors, Infocom Games, Spread Sheets, & Utilities. Some CP/M-86 Public Domain. Over 300 Disk Formats. No extra charge for yours! Disk copying services including DOS, Apple, Mac, Atari & Amiga. SASE for free information. 30 page catalog $1.00.

Elliam Associates
Box 2664-T
Atascadera, CA 93423
(805) 466-8440

The Computer Corner

by Bill Kibler

Another article with more on a hot topic (at least for me)—programming. It occurred to me I hadn't explained what or how you can determine if your program is well structured. So let's see if we can't give you some pointers into better code generation and programmer management.

Defining The Problem

The reason I have been so much involved lately with what constitutes good and bad programming is my job. I work for a company who has 10 year old code and can not maintain it. I know that they are not unusual in any way. Lots of companies are finding themselves in this situation. There are a number of mistakes they have all made and as we will see, some ways they could have prevented those problems.

Let's talk about my current problems and how they got there. This is 68000 assembly language code. In many cases this was the first time any of these programmers had written assembly language code. They knew only high level structures and imported them to the code whenever possible. They have used macros in order to create the higher level structures they thought were needed. Many of the programmers felt only they were capable of writing good code. Personality problems added to the way code was generated. Some of the original employees are still here and I can see how they are still messing up the other programmers by forcing their concepts of good programming on them.

Before I explain how to test good style from bad, what is bad programming. I have 10 years worth of code that is mostly unreadable. It runs on for pages and appears to do nothing. They have used macros which are not transportable to other assemblers. The have gone so far as to write their own computer language. To make it worse they have incorporated macros to talk to their own language. They use those special macros even when the code is going into ROMs that do not interface to their code. Since many of the programmers have left the company, there are large sections of code in which we have no person to ask to find out what they are suppose to do. Documentation is almost non existent. There is not a single document that explains how the actual code works, which modules talk to which modules, what method of passing data is used, or what the overall functionality of the program is. We can't rewrite the code because we do not know what the code actually does.

How Not to Get There...

The company got themselves in this problem for a number of reasons. Early on, the managers should have been in charge and were not. By this I mean, taking control from your programmers. As a manager you need to know where the company is going, what the long term objectives of the product are, need clear functional specifications of the product, and take charge of the project. A new manager here, has his programmers write specifications of the proposed code before any code is written. This is a good start.

Where he and most places fall apart is checking on the actual code written. Nobody here reviews the code once it is written. If it works, it becomes a product. That is how 10 years of really bad, but working code, has been keeping the company going. Now they want to keep their customers happy and are having real problems fixing minor bugs and adding simple upgrades. What was needed all along was a review committee on written code.

If you are a single programmer working on a project, you will know every bit of code. If however you are part of a group working together on a large project, your code must fit in with others and be readable by them as well. The only way to guarantee that is to have the group review that code. For small groups everybody would get involved. For larger groups, committees of three or four would do. The main task for the manager is making sure the reviewers do it properly.

Some simple guidelines are needed, and it is up to the manager to take charge and enforce those guidelines. Most of the code I see does work very well. The problems come when that code is stuck into, or as the case may be not into modules. "Spaghetti" code typically has good sections, however those sections need to be "factored" out. Factoring is breaking the code into smaller modules so that the same functions are not repeated hundreds of times with new code.

What I have used for myself is a list of questions that need to be asked for every program you write or are involved with. The reviewers need to continuously ask these questions and use them as guide lines. They are:

1) What will be needed if this code is ported over to another operating system, hardware platform, assembler, or programming style.
2) What will happen if new features are needed, can they be added easily.
3) Does the structure lend itself to easy maintenance, how are minor changes made, can bugs be found and corrected easily.
4) Is the structure clearly defined and documented so we can work on the project two years from now without major problems.
5) What happens if the lead programmer dies, moves, or gets sick.

The last one is my favorite and usually causes the most problems. It becomes a problem when they answer back that "I will get it done after .... ", because that after never happens. As a manager you can not ignore these five concepts, doing so is a guarantee that your code will not be portable, maintainable, or repairable. A bottom line here is setting standards for your programming operation.

Those standards involve how the code is written, not only in what language used, but style, version tracking, documentation, flow charting, testing, and generation. Clearly defined standards for these aspects

(Continued on page 39)